Professional
haXe and Neko

Part I: The Core Language
Chapter 1: Introducing haXe .................................................................3
Chapter 2: Installing and Using haXe and Neko ......................................11
Chapter 3: Learning the Basics .............................................................27
Chapter 4: Controlling the Flow of Information ....................................73
Chapter 5: Delving Into Object-Oriented Programming .......................101
Chapter 6: Organizing Your Code .........................................................139
Chapter 7: When Things Go Wrong .....................................................179

Part II: Server Side, JavaScript, and Flash; Oh My!
Chapter 8: Cross Platform Tools ..........................................................203
Chapter 9: Building Websites with HaXe ..............................................225
Chapter 10: Separating Design Using Templates ...................................251
Chapter 11: Performing Server-Side Trickery .......................................271
Chapter 12: Building Interactive Content with Flash .........................315
Chapter 13: Replacing the Need for an IDE .........................................363
Chapter 14: More Interactive Content with JavaScript .....................387
Chapter 15: Putting It All Together with haXe Remoting ....................425

Part III: Extending the Possibilities
Chapter 16: haXe Advanced Topics ....................................................455
Chapter 17: Desktop Applications with Neko .......................................479
Chapter 18: Desktop Flash with SWHX ..............................................509
Chapter 19: Multimedia with Neko .....................................................531
Chapter 20: Extending haXe with C/C++ .............................................555
Appendix A: Semantic HTML ...............................................................577
Index ......................................................................................................603
Professional
haXe and Neko

Franco Ponticelli
Lee McColl-Sylvester

Wiley Publishing, Inc.

www.allitebooks.com
To Cristina and Gabriel, the meaning of my life
—Franco Ponticelli

For Jay, Tabitha, Reuben, and Bradley
“The best feelings are those that have no words to describe them . . .”
—Lee McColl-Sylvester
About the Authors

Franco Ponticelli is an experienced developer and solution architect. An Architecture Graduate with specialization in Industrial Design, he performed many different activities in the Information Technology area from 3D Computer Graphics to hard-core software development. In his continual research for the perfect development environment, he found haXe and fell in love.

Franco is currently a self-employed developer and you can reach him on his personal website www.weblob.net.

Lee McColl-Sylvester is an expert ActionScript developer as well as a seasoned master in systems integrations. A student in Visual Communications, he evolved his career specializing in advanced graphical interface development, as well as information management systems, database architecture, and hardware communications implementation. An inventor at heart, Lee discovered haXe while tinkering in the Open Source Flash arena.

Lee is self-employed and can be reached through the haXe mailing list, or his website www.designrealm.co.uk.
Acknowledgments

My first and foremost thanks go to Gisella, without her support, it would have been impossible for me to realize this work.

Special thanks goes to co-author Lee McColl-Sylvester, who involved me in this project in the first place. I sincerely hope to work with him again because of his professionalism, kindness, and competence.

Many thanks also to Nicolas Canasse, the haXe author, whose expertise and talent are incomparable. Many thanks to all of the members of the haXe mailing list too, they are always ready to give you an answer to every doubt or concern you can have about the haXe experience. You can reach the mailing list at this address: http://lists.motion-twin.com/mailman/listinfo/haxe.

A big hug goes to Ed Connor, Daniel Fischer, and Chris Webb whose help has been fundamental. I really appreciated their aid and guidance.

Finally, I have to thank my family and friends who always support my choices unconditionally.

Francesco Ponticelli

From the deepest of my heart, I’d like to thank my wife, Jay, for her patience and understanding at a time without reason. Also, my children, Tabitha, Reuben and Bradley, who never fail to make me smile.

I’d like to thank Francesco Ponticelli for being a fantastic co-author. This book would never have been possible without his support, enthusiasm, dedication, and friendship. I’d also like to thank Nicolas Canasse for his help and advice, and for the haXe language and Neko platform.

A big thank you to Daniel Fischer for his excellent efforts as a Technical Editor. Also, Ed Connor and Chris Webb for their understanding, motivation and tireless support as editors.

Thanks to Ritchie “Blackdog” Turner, Grant “Resident Alien” Davies, and Kevin “The Masked Coder” Murphy for their advice and friendship.

More thanks to Wesley Russell Blampied for giving me the idea to write this book. Also, Glynn Hayward, James Jackson and Nigel Britton of Complete Control UK Limited for their support, kindness, generosity, and understanding. I did it, guys!

Thanks to my old team at Element 78 Ltd for joining me on many years of Flash coding. It was always a pleasure working with them, and I miss every minute of it.

Finally, thanks to the whole haXe and Neko mailing lists for their dedication to a fantastic family of development tools and to those who learn a little something from the words contained in this book.

I thank you . . .

Lee McColl-Sylvester
Foreword

Writing a book about haXe is not an easy task. Not because there are complicated things to explain: haXe is a new programming language. Its features are mostly coming from classic object-oriented languages such as Java, with some having been taken from more dynamic Scripting languages, and some others from Functional languages. This mix makes haXe a unique language, but all these features one by one are pretty easy to explain and understand.

What makes a haXe book difficult to write is the amount of possibilities that haXe opens. By being able to target three different platforms — JavaScript, Flash, and Neko — haXe opens a lot of doors for the web developer. And because Neko is an extensible virtual machine, using haXe opens more doors behind pure web development, such as real-time servers, console and desktop applications, and all the things that a modern programming language can do.

This is the main reason why the haXe community is so diverse. People are coming from different languages with different goals. Some are tired of JavaScript and want to use a modern language such as haXe to speed up web development, some want to develop Flash content without relying on proprietary tools, some want to experiment with scalable technologies for website development and database handling, some even want to develop desktop OpenGL games using haXe!

This diversity is a gift. By combining different technologies, it's now possible to create things that were yesterday very difficult and costly to realize because of the difficulty of learning and integrating these different technologies together. By using haXe everywhere, it's now possible to quickly develop multitechnologies applications, such as highly interactive Flash-JS-Server websites. By using haXe everywhere, people with knowledge in one of these platforms can now leverage their skills and use it to develop for other platforms as well, as soon as they need it. This is the biggest success of haXe: opening doors for developers so that different technologies and platforms are no longer a limitation for creativity.

But that's also a problem when it comes to presenting haXe clearly: explaining its different possible applications already takes a few lines. Entering into details and showing examples of all the possibilities that haXe offers is very time consuming and requires time to learn and experiment with a lot of technologies. While this book mostly will focus on web development: web servers, databases, Flash and JavaScript, it will also give you the keys for other technologies, such as real-time servers, that will open as many doors.

It's for these reasons that I really want to thank the book's authors: Lee McColl-Sylvester and Franco Ponticelli, for their work on this book, which I sincerely think will provide you with a very good introduction, samples, and detailed explanations of the most important aspects of the haXe Programming Language. I thank them for their hard work in putting all these pieces together.
Foreword

Last thing: If I can give you a tip for reading this book, it’s not to try to read every single page of it before starting using haXe. First focus on the basics, then read the parts that you are more familiar with or that you are most eager to learn, then experiment, modify, and try it by yourself. If you have questions, contact the haXe mailing list; there will be people there ready to help you. Don’t read this book like you would read a novel, but more like an encyclopedia.

Thank you, and don’t forget to have fun,

Nicolas Cannasse
haXe creator
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgments</td>
<td>xi</td>
</tr>
<tr>
<td>Foreword</td>
<td>xiii</td>
</tr>
<tr>
<td>Introduction</td>
<td>xxv</td>
</tr>
<tr>
<td><strong>Part I: The Core Language</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Chapter 1: Introducing haXe</strong></td>
<td>3</td>
</tr>
<tr>
<td>A Problem with Internet Development</td>
<td>3</td>
</tr>
<tr>
<td>What Is haXe and Why Was It Created?</td>
<td>4</td>
</tr>
<tr>
<td>The haXe Compiler</td>
<td>5</td>
</tr>
<tr>
<td>The haXe Language</td>
<td>7</td>
</tr>
<tr>
<td>How Does haXe Work?</td>
<td>8</td>
</tr>
<tr>
<td>So What Is Neko?</td>
<td>8</td>
</tr>
<tr>
<td>haXe and Neko Requirements</td>
<td>9</td>
</tr>
<tr>
<td>Summary</td>
<td>10</td>
</tr>
<tr>
<td><strong>Chapter 2: Installing and Using haXe and Neko</strong></td>
<td>11</td>
</tr>
<tr>
<td>Installing haXe</td>
<td>11</td>
</tr>
<tr>
<td>Using the haXe Automatic Installer</td>
<td>12</td>
</tr>
<tr>
<td>Manually Installing haXe on Windows</td>
<td>12</td>
</tr>
<tr>
<td>Manually Installing haXe on Linux</td>
<td>13</td>
</tr>
<tr>
<td>Installing Neko</td>
<td>14</td>
</tr>
<tr>
<td>Manually Installing Neko on Windows</td>
<td>14</td>
</tr>
<tr>
<td>Installing Neko on Linux</td>
<td>16</td>
</tr>
<tr>
<td>Hello World! in Triplicate</td>
<td>17</td>
</tr>
<tr>
<td>Compiling Your First haXe Application</td>
<td>17</td>
</tr>
<tr>
<td>How Your Program Was Compiled</td>
<td>18</td>
</tr>
<tr>
<td>HXML Compiler Files</td>
<td>19</td>
</tr>
<tr>
<td>Compiling to Neko</td>
<td>19</td>
</tr>
<tr>
<td>Compiling to JavaScript</td>
<td>21</td>
</tr>
<tr>
<td>Program Structure</td>
<td>22</td>
</tr>
<tr>
<td>Compiler Switches</td>
<td>23</td>
</tr>
<tr>
<td>Summary</td>
<td>25</td>
</tr>
</tbody>
</table>
## Contents

### Chapter 3: Learning the Basics 27

- **The haXe Hierarchy** 27
- **The Standard Data Types** 28
- **Variables** 28
  - Type Inference 30
  - Constant Variables 31
- **The Simple Value Types** 31
  - Floating-Point Numbers 32
  - Integers 32
  - Which Number Type? 33
  - Booleans 33
  - Strings 33
- **Abstract Types** 35
  - Void and Null 35
  - The Dynamic Type 35
  - The unknown Type 36
- **Bypassing Static Typing with untyped** 36
- **Commenting Your Code** 37
  - Block Commenting 37
  - Line Commenting 37
- **Converting Data Through Casting** 38
  - The cast Function 38
  - Simple Type Conversion Functions 39
  - Comparing the Type of a Value 40
- **Using Arrays in haXe** 40
  - Arrays 41
  - Lists 46
  - Hash Tables 48
- **Using Dates in haXe** 50
  - Creating a Date Object 51
  - Retrieving the Components of a Date 52
  - The DateTools Class 53
- **Manipulating Data** 55
  - Operators 55
  - The Math Class 63
  - Using String Functions 66
- **Summary** 72
## Contents

### Chapter 4: Controlling the Flow of Information 73

**Pushing Data Around** 73

**Conditional Statements** 74
- if Statements 74
- switch Statements 77
- Returning Values from Conditional Statements 78

**Loops** 79
- The while Loop 80
- The for Loop 81
- Looping Over Collections 82
- Break and Continue 84

**Functions** 85
- Class Functions 85
- Local Functions 90
- The Lambda Class 96

**Summary** 99

### Chapter 5: Delving Into Object-Oriented Programming 101

**Classes and Objects** 102
- Instance Fields 104
- Static Fields 114

**Understanding Inheritance** 115
- Super 117
- Functions Override 117
- Constructor Override 118
toString() 119
- Abstract Classes and Abstract Methods 119
- Static Fields, Instance Variables, and Inheritance 120
- Inheritance Roles 120

**Using Interfaces** 121

**Advanced Classes and Objects Features** 122
- Implementing Classes 123
- Type Parameters 123
- Anonymous Objects 126
- Implementing Dynamic 127
- Typedef 128
- Extensions 133

**Enum** 134
- Constructor Arguments 136

**Summary** 138

xvii
# Contents

## Chapter 6: Organizing Your Code

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building Reusable Code</td>
<td>139</td>
</tr>
<tr>
<td>Using Packages</td>
<td>140</td>
</tr>
<tr>
<td>Declaring a Package</td>
<td>140</td>
</tr>
<tr>
<td>Implicit Import</td>
<td>141</td>
</tr>
<tr>
<td>Explicit Import</td>
<td>142</td>
</tr>
<tr>
<td>Enum and Packages</td>
<td>143</td>
</tr>
<tr>
<td>Type Lookup Sequence</td>
<td>144</td>
</tr>
<tr>
<td>Importing an Entire Package</td>
<td>144</td>
</tr>
<tr>
<td>Importing Libraries</td>
<td>145</td>
</tr>
<tr>
<td>haXe Standard Library</td>
<td>145</td>
</tr>
<tr>
<td>haxelib Libraries</td>
<td>146</td>
</tr>
<tr>
<td>Libraries from Other Projects</td>
<td>150</td>
</tr>
<tr>
<td>External Libraries</td>
<td>150</td>
</tr>
<tr>
<td>Using Resources</td>
<td>163</td>
</tr>
<tr>
<td>Documenting Your Code</td>
<td>166</td>
</tr>
<tr>
<td>Offline Documentation</td>
<td>167</td>
</tr>
<tr>
<td>Online Documentation</td>
<td>169</td>
</tr>
<tr>
<td>Unit Testing</td>
<td>170</td>
</tr>
<tr>
<td>The haxe.unit Package</td>
<td>171</td>
</tr>
<tr>
<td>Writing Tests</td>
<td>172</td>
</tr>
<tr>
<td>Summary</td>
<td>177</td>
</tr>
</tbody>
</table>

## Chapter 7: When Things Go Wrong

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Trace Function</td>
<td>179</td>
</tr>
<tr>
<td>Trace Output</td>
<td>180</td>
</tr>
<tr>
<td>haXe Trace versus ActionScript Trace</td>
<td>181</td>
</tr>
<tr>
<td>Exceptions</td>
<td>185</td>
</tr>
<tr>
<td>Dealing with Exceptions</td>
<td>186</td>
</tr>
<tr>
<td>The CallStack and ExceptionStack</td>
<td>191</td>
</tr>
<tr>
<td>The Exception Master Class</td>
<td>194</td>
</tr>
<tr>
<td>Creating a Complete Exception Class</td>
<td>194</td>
</tr>
<tr>
<td>The Exception Class Code</td>
<td>197</td>
</tr>
<tr>
<td>Summary</td>
<td>199</td>
</tr>
</tbody>
</table>

## Part II: Server Side, JavaScript, and Flash; Oh My!

## Chapter 8: Cross Platform Tools

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML</td>
<td>203</td>
</tr>
<tr>
<td>XML Anatomy</td>
<td>203</td>
</tr>
<tr>
<td>haXe XML API</td>
<td>206</td>
</tr>
</tbody>
</table>
### Contents

#### Regular Expression 210
- The EReg Class 210
- Patterns 214

#### Timer 222
- Delayed Action 223
- Queued Actions 223

#### MD5 224

#### Summary 224

### Chapter 9: Building Websites with HaXe 225

#### Introduction to Web Development 225
- What Is a Web Server? 226
- Distributing Contents Using a Web Server 226
- An HTML Crash Course 228
- How haXe Differs from HTML 230

#### The NekoTools Web Server 231

#### Installing mod_neko For Apache 232
- Installing mod_neko on Windows XP 232
- Installing mod_neko on Linux (Ubuntu) 233

#### Your First Website with haXe 234
- Using Neko as a Page Controller 235

#### The neko.Web Class 236
- Using Neko as a Front Controller 238

#### Summary 250

### Chapter 10: Separating Design Using Templates 251

#### What Are Templates? 251

#### The Template Class 252
- The Template Syntax 253

#### Using Resources 257
- When to Use Code in Templates? 258

#### Templo for Server-Side Templates 258
- Installing Templo 259
- Using Templo 259
- Differences in Expressions Between haxe.Template and mtwin.Templo 261
- Attr Expressions 262
- Raw Expressions 262
- Logical Expressions 263
- Loop Expressions 263
- set, fill, and use Expressions 264

www.allitebooks.com
## Contents

Using Macros with Templo 266  
Compiling Templates Manually 268  
**Summary** 269  

### Chapter 11: Performing Server-Side Trickery 271

**Introducing Neko** 271  
The Neko Virtual Machine 272  
The Neko Standard Libraries 272  
**Working with Databases** 273  
Choosing a Database 273  
Connecting to the Database 276  
The Connection Class 277  
The ResultSet Class 283  
**Using the SPOD System** 286  
Getting Started 287  
Simple Use of the SPOD System 287  
Table Mapping with neko.db.Object 289  
The neko.db.Manager Class 291  
Adding Table Relations 295  
**Working with Files** 301  
Traversing Files and Directories 301  
Working with Files and Streams 304  
Creating a Logger Class 311  
**Summary** 314  

### Chapter 12: Building Interactive Content with Flash 315

**The Flash Library** 315  
The Flash Movie 316  
The Movie Clip 319  
Text 322  
Multimedia Contents 323  
Loading Values 325  
Events 328  
Events over Time 331  
The Drawing API 337  
Extending the MovieClip Class 342  
**Flash to haXe** 344  
Data Types 344  
Global Constants and Special Values 346  
Top-level Functions 347
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operators</td>
<td>352</td>
</tr>
<tr>
<td>Statements, Structures, and Keywords</td>
<td>354</td>
</tr>
<tr>
<td>Global Properties</td>
<td>356</td>
</tr>
<tr>
<td>Top-level Classes Replaced by haXe-Specific Types</td>
<td>357</td>
</tr>
<tr>
<td>Flash API Classes</td>
<td>359</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>362</td>
</tr>
</tbody>
</table>

#### Chapter 13: Replacing the Need for an IDE

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assets with SWFMill</td>
<td>364</td>
</tr>
<tr>
<td>Creating an Image Movie Clip</td>
<td>365</td>
</tr>
<tr>
<td>Embedding Fonts</td>
<td>368</td>
</tr>
<tr>
<td><strong>User Interfaces</strong></td>
<td>369</td>
</tr>
<tr>
<td>How to Implement an UI</td>
<td>369</td>
</tr>
<tr>
<td>Events</td>
<td>371</td>
</tr>
<tr>
<td>Views, Controls, Containers, and Elements</td>
<td>373</td>
</tr>
<tr>
<td>Element</td>
<td>373</td>
</tr>
<tr>
<td>Controls</td>
<td>375</td>
</tr>
<tr>
<td>Containers</td>
<td>376</td>
</tr>
<tr>
<td>The Flash 9 Implementation</td>
<td>381</td>
</tr>
<tr>
<td>A Complete Example: the Tree Editor</td>
<td>381</td>
</tr>
<tr>
<td>Conclusions</td>
<td>384</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>385</td>
</tr>
</tbody>
</table>

#### Chapter 14: More Interactive Content with JavaScript

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Content in JavaScript?</td>
<td>387</td>
</tr>
<tr>
<td>The Web Developer Tools</td>
<td>389</td>
</tr>
<tr>
<td>Trace to Firebug</td>
<td>389</td>
</tr>
<tr>
<td><strong>JavaScript as haXe Target</strong></td>
<td>390</td>
</tr>
<tr>
<td>The JavaScript Library</td>
<td>391</td>
</tr>
<tr>
<td><strong>Manipulating the Page with JavaScript</strong></td>
<td>396</td>
</tr>
<tr>
<td>The OnLoad Problem</td>
<td>396</td>
</tr>
<tr>
<td>Advanced JavaScript Libraries</td>
<td>399</td>
</tr>
<tr>
<td>Traversing HTML with JavaScript</td>
<td>400</td>
</tr>
<tr>
<td><strong>What Is AJAX?</strong></td>
<td>414</td>
</tr>
<tr>
<td>Removing the Myths Around AJAX Complexity</td>
<td>415</td>
</tr>
<tr>
<td>Updating a Panel Content Using AJAX</td>
<td>416</td>
</tr>
<tr>
<td>Building an Auto-Complete Control</td>
<td>417</td>
</tr>
<tr>
<td><strong>Advantages of haXe/JS Development</strong></td>
<td>422</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>423</td>
</tr>
</tbody>
</table>
Contents

Chapter 15: Putting It All Together with haXe Remoting 425

What Is haXe Remoting? 425
  Remote Connection and Local Connection 426
  Synchronous and Asynchronous Communications 426
  Pull or Push? 426
The haXe Remoting API 427
  Connection 428
  AsyncConnection 432
  FlashJsConnection 435
  LocalConnection 435
  SocketConnection 436
  NekoSocketConnection 436
  AsyncDebugConnection, DelayedConnection, and AsyncAdapter 437
  SocketProtocol 437
Proxy Objects 437
TicTacToe Game 443
Summary 452

Part III: Extending the Possibilities

Chapter 16: haXe Advanced Topics 455

XML 455
  Fast 455
  Check 458
  XML Proxy 463
Reflection API 464
  The Reflect Class 464
  The Type Class 467
  Runtime Type Information (RTTI) 470
Serialization 473
haXe Magic 475
Summary 478

Chapter 17: Desktop Applications with Neko 479

Why Use Neko for Desktop Applications? 479
Creating a Neko Executable 480
Graphical User Interface Libraries 480
  ScreenWeaver HX 480
  Xinf: Xinf Is Not Flash 481
Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>nGui</td>
<td>481</td>
</tr>
<tr>
<td>hxGtk</td>
<td>481</td>
</tr>
<tr>
<td><strong>Installing the Required Library</strong></td>
<td>481</td>
</tr>
<tr>
<td>The nGui Library</td>
<td>482</td>
</tr>
<tr>
<td>The nGui Manager Class</td>
<td>482</td>
</tr>
<tr>
<td>Working with Static Controls</td>
<td>484</td>
</tr>
<tr>
<td>The Control Class</td>
<td>485</td>
</tr>
<tr>
<td>Container Classes</td>
<td>487</td>
</tr>
<tr>
<td>Working with Interactive Controls and Events</td>
<td>488</td>
</tr>
<tr>
<td>Working with Menus</td>
<td>496</td>
</tr>
<tr>
<td>The hxGtk Library</td>
<td>499</td>
</tr>
<tr>
<td>The hxGtk Function Proxy</td>
<td>499</td>
</tr>
<tr>
<td>The Gtk Class</td>
<td>501</td>
</tr>
<tr>
<td>hxGtk Events</td>
<td>502</td>
</tr>
<tr>
<td><strong>Rapid hxGtk GUI Development with Glade</strong></td>
<td>503</td>
</tr>
<tr>
<td>The Glade IDE</td>
<td>504</td>
</tr>
<tr>
<td>The Glade Class</td>
<td>505</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>506</td>
</tr>
</tbody>
</table>

Chapter 18: Desktop Flash with SWHX                                    509

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why SWHX?</td>
<td>510</td>
</tr>
<tr>
<td>Installing SWHX</td>
<td>511</td>
</tr>
<tr>
<td>Getting Started with SWHX</td>
<td>511</td>
</tr>
<tr>
<td>Working with Windows</td>
<td>513</td>
</tr>
<tr>
<td>Window Decoration Flags</td>
<td>513</td>
</tr>
<tr>
<td>Window Properties</td>
<td>514</td>
</tr>
<tr>
<td>Window Events</td>
<td>515</td>
</tr>
<tr>
<td><strong>Communication Between Flash and Neko</strong></td>
<td>517</td>
</tr>
<tr>
<td>Asynchronous Communication</td>
<td>518</td>
</tr>
<tr>
<td>Synchronous Communication in haXe</td>
<td>520</td>
</tr>
<tr>
<td>Synchronous Communication in ActionScript</td>
<td>523</td>
</tr>
<tr>
<td><strong>Using SysTools</strong></td>
<td>525</td>
</tr>
<tr>
<td>Creating a System Tray Icon</td>
<td>526</td>
</tr>
<tr>
<td>Creating Menus</td>
<td>526</td>
</tr>
<tr>
<td>Using Dialogs</td>
<td>529</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>530</td>
</tr>
</tbody>
</table>

Chapter 19: Multimedia with Neko                                       531

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why Use Neko for Games Development?</td>
<td>532</td>
</tr>
<tr>
<td>Neko’s NME is Your Friend</td>
<td>532</td>
</tr>
</tbody>
</table>

Summary
Contents

Getting Started with NME 533
The Manager Class 534
Writing Text to the Display 537
Working with Surfaces 537
Keeping Time with Timers 546
The Wonderful World of Sprites 546
Interaction Through Events 549
Summary 553

Chapter 20: Extending haXe with C/C++ 555

Neko Libraries 555
Setting Up the Compiler 556
A Simple Hello World Example 556
Building the ndll Library in C 556
Building the haXe Script 557
The Post-Mortem 557
Neko Value Types in C/C++ 558
The Neko value Struct 558
Preparing Data for Transit from haXe to C/C++ 559
Restructuring Data in haXe from C/C++ 560
Neko Type Checking in C/C++ 560
Passing C/C++ Values to Neko 562
Abstract Types 563
Object Types 565
Arrays 567
Functions 568
Managing Neko Primitives 570
Garbage Collection 571
alloc_root() 571
free_root() 572
alloc() 572
Garbage Collecting Abstract Values 572
Error Handling 573
val_throw() and val_rethrow() 573
failure() 573
bfailure() 574
val_print() 574
neko_error() 574
Summary 574

Appendix A: Semantic HTML 577

Index 603

xxiv
Introduction

haXe is a language, a compiler, and a framework of classes designed to empower the developer to write clean code targeted at multiple platforms on multiple operating systems. Based on the famous open source MTASC ActionScript compiler and written by the author of MTASC, haXe opens up the world of applications development while reducing the learning curve and minimizing roadblocks.

Neko is a cross-platform virtual machine, and one of the targets of the haXe compiler. It is lightweight, as well as incredibly fast, flexible, extensible, and reliable. It can be embedded or extended and requires a footprint so small as to appear almost invisible.

Professional haXe and Neko aims to target these two tools and the relationship between them. Within the pages of this book, you will learn how to command the awesome power of the Flash platform, how to develop server-side web applications with ultimate functionality, how to manipulate client web browsers with unmatched JavaScript, and how to combine all three into unprecedented desktop applications. Professional haXe and Neko will take you on a ride through the leading technologies in the industry and show you how to bend them to your will. Plus, you’ll have fun doing it.

Written by two guys who love haXe and Neko with a passion, this book aims to reveal how haXe and Neko can render all other web-related technologies obsolete. From the very first chapter, you’ll be taken on a detailed journey through the capabilities of these technologies and you’ll stretch them to the limit. You’ll learn how to master graphical user interface development for a multitude of purposes, program reusable code modules using a true ECMAScript language, tackle the browser DOM with object-oriented JavaScript, and more.

Professional haXe and Neko is the first book to cover these fantastic new technologies, and may possibly be the last book you ever need.

Whom This Book Is For

This book is for existing web developers who wish to work with a unified environment and reduce the number of languages required to produce a single software solution. You don’t have to be a seasoned programmer to understand the concepts within this book, though if you have already programmed in ActionScript, Java, or some other ECMA standard language, then things will come to you a lot faster. As strange as it sounds, most developers who come to haXe are those who do not want to learn a new language. Many ActionScript 2.0 developers, for example, may choose not to progress to ActionScript 3.0, simply because it follows a slightly different syntax. haXe, however, maintains a singular language, regardless of target platform, and therefore preserves the developer’s language experience.

What This Book Covers

At the time of writing, the versions of the software covered by this book are haXe version 1.15 and Neko version 1.6.0. Both haXe and Neko are in constant evolution as there is always a constant activity in the various community meeting places, and it would not be surprising to see two or more minor releases
Introduction

appear between the book’s completion and the publishing date. However, new releases do not usually
break backward compatibility, so you need not worry about information within this publication being
out of date. If you are in any doubt about the content of this book, just check the haXe website, which
will list the features added and direct you to how these changes have affected the book.

This book describes the use of haXe to target the generation of output for the following platforms: Neko,
Flash Player versions 6 to 9, and JavaScript. Within the pages of this book, you will learn everything you
need to produce first class applications for any of these environments.

How This Book Is Structured

Professional haXe and Neko is split into three parts, depending on the content being delivered.

The first part of the book is dedicated to the basics of the haXe language, providing a thorough detail of
the language structure, standard libraries, and programming best practices. An experienced haXe
programmer can skip this section entirely while a seasoned programmer learning haXe for the first time
can scan it rapidly and take a deeper look at both the code examples and the tables detailing the classes
and commands required to develop in haXe.

The second part of the book covers practical aspects of haXe, such as producing Flash movies and
dynamic content for websites. This part is of interest to programmers of any skill level. It also contains
references to the vast majority of platform-specific classes defined in the standard library, which is the
core framework that comes with the haXe base installation. Some external libraries from the haXeLib
repository are also described.

The third part of the book is dedicated to the more advanced developer who wants to push the haXe
language to its limit, and seeks to extend haXe with existing libraries or use haXe outside of the
conventional web environment.

What You Need to Use This Book

Developing with haXe and Neko is a cheap affair, as all the tools described in the book are open source
and available for free on the Internet. You will find there aren’t any particular requirements, and both
haXe and Neko will operate on Windows, Linux, and both types of Apple Macintosh. All you need to get
started with haXe and Neko is to follow the step-by-step procedures as outlined in Chapter 2.

The use of a more sophisticated editor can improve your productivity dramatically. FlashDevelop
(www.flashdevelop.org/) is a very good alternative to tools such as Notepad in the Windows
environment, and can certainly save a lot of headaches when you first start out. The current version of
FlashDevelop supports haXe directly out of the box, while older versions can install the haxefd plug-in
from the haxelib repository. An alternative development environment for the Linux/Mac OS X operating
systems is the promising haxeDevelop (http://code.google.com/p/haxedevelop/). As more and
more editors support the haXe language, they will begin to appear on the haXe mailing list, so be sure to
keep an eye open.
Introduction

Conventions

To help you get the most from the text and keep track of what’s happening, we’ve used a number of conventions throughout the book.

Boxes such as this one hold important, not-to-be-forgotten information that is directly relevant to the surrounding text.

Notes, tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.

As for styles in the text:

- We highlight new terms and important words when we introduce them.
- We show filenames, URLs, and code within the text like so: `Reflect.field()`.
- We present code in two different ways:
  - We use a monofont type with no highlighting for most code examples.
  - We use gray highlighting to emphasize code that’s particularly important in the present context.

Commands that must be executed in the DOS prompt (for Windows) or in the terminal/console (for Mac OS X and Linux) are shown prefixed with an angular bracket followed by a whitespace.

> haxe build.hxml

Source Code

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at www.wrox.com. Once at the site, simply locate the book’s title (either by using the Search box or by using one of the title lists) and click the Download Code link on the book’s detail page to obtain all the source code for the book.

Because many books have similar titles, you may find it easiest to search by ISBN; this book’s ISBN is 978-0-470-12213-6.

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at www.wrox.com/dynamic/books/download.aspx to see the code available for this book and all other Wrox books.
Introduction

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, such as a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and at the same time you will be helping us provide even higher quality information.

To find the errata page for this book, go to www.wrox.com and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list including links to each book’s errata is also available at www.wrox.com/misc-pages/booklist.shtml.

If you don’t spot “your” error on the Book Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We’ll check the information and, if appropriate, post a message to the book’s errata page and fix the problem in subsequent editions of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at p2p.wrox.com. The forums are a web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At http://p2p.wrox.com you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to p2p.wrox.com and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.
Part I

The Core Language

Chapter 1: Introducing haXe
Chapter 2: Installing and Using haXe and Neko
Chapter 3: Learning the Basics
Chapter 4: Controlling the Flow of Information
Chapter 5: Delving Into Object-Oriented Programming
Chapter 6: Organizing Your Code
Chapter 7: When Things Go Wrong
Introducing haXe

The Internet is a fantastic invention. As developers, we determine its future and shape its usefulness through our own developments. This book will hopefully help to refine this path, as we seek to make the Internet ever more perfect.

In this chapter, you will acquire a broad perspective of:

- Current issues with developing for the Internet.
- What are haXe and Neko?
- How haXe and Neko can alleviate issues with Internet development.

A Problem with Internet Development

The IT developer, regardless of skills and title, can often be perceived as sitting somewhere along a scale depicting the transition from the creative to the logical. On the creative end of the scale, you might see those who design website layouts, digital illustrations, animations or presentations, while on the purely logical side, you might see the database administrators or server-side architects. Of course, most people located under the developer umbrella sit somewhere in between and cover more than a little speck along this line.

The term *developer* tends to describe those who deal with some form of programming language. The highly creative ones might use ActionScript or JavaScript to accomplish their development needs, while the very logical may be restricted to using PL/SQL or C, perhaps. Yet, regardless of which languages or development skills you claim to have, the diversity of the tasks you are charged to accomplish require that you use numerous technologies.

As an example, if you were to look at the technologies that may be required for the development of a web application, you might decide to use the services of a Flash designer, an HTML and
Part I: The Core Language

JavaScript developer, a server-side applications developer, and a database architect. This will certainly cover a lot of the requirements for many web-based projects, and of course, many developers can handle more than one of these points very well. However, software development demands that the various layers that constitute the final application will function and communicate well with each other and you cannot always guarantee that the members of a development team will understand the roles and technologies of the other team members with the depth required to facilitate this level of reliability. Similarly, when a single developer is charged with providing his or her services to build such an application, it is likely they would much prefer to spend the time they have been paid for in developing the application as opposed to having to learn the various new technologies required to complete the project.

This is where haXe makes its entrance.

What Is haXe and Why Was It Created?

Over recent years, we have seen the introduction of numerous new technologies and languages that make applications development a little bit more exciting. Languages such as Python and Ruby have appeared that, while not necessarily providing groundbreaking features, lend some support to providing alternative ways to producing software. Ruby on Rails, or RoR, is one such feature that has enabled developers to produce web-based applications with remarkable speed and ease, making the development of common application functionality less painful.

This is a great move in the right direction, and many developers have since made the transition to such languages to further lessen the strain that was ever present with languages such as C++ or Java, but it doesn’t provide a be-all-and-end-all for applications development as it still means having to learn a new technology. What’s more, despite the surge of new technologies that have been appearing, you can see that they are only really of benefit to desktop or server-side web development, while the client side for web applications remain firm with Flash, HTML, JavaScript, and VBScript, for the simple reason that older browser technologies have a greater coverage on users’ machines. This means that, while we now have a larger choice of tools for creating our applications, we are perhaps further segregating our development team members and forcing possible rifts into our software stability and production timescales.

So why are we suffering these issues and what can be done to dispose of these rifts we keep making for ourselves while continuing to embrace technology developments? The issues here are not about how such technologies interact, or about the functionality they provide, but about how we develop for them.

Think about that for a second. If humans, like technology, needed to interact with one another, we would use an agreed form of communication; written English in the case of this book. If we were to seek employment in a country whose entire population didn’t speak our language, we are sure this would hinder our progress very much, as it would be hard enough to even convey that we were seeking employment, let alone be capable of performing many of the available work positions on offer. Yet, in respect of many programming languages and technologies available, they often communicate very well with each other using processes of their own, but the way in which we interact with them can be very diverse.
Chapter 1: Introducing haXe

When new languages are created, the syntax they support is often chosen to reflect the flavor of programmer the language is likely to attract, the features the language provides, the structure of the compiler, and the general taste of the language’s author, to name a few. However, a compiler is often just a clever program that analyzes your code and converts it to a more machine readable syntax, so it is just as plausible to create a compiler that can compile from one language into another. This is exactly what Nicolas Cannasse thought, and what prompted him to create the haXe compiler.

The haXe Compiler

The original purpose of the haXe compiler was to encompass several web technologies under a single language. Supporting Flash, JavaScript and Neko, web developers can use the haXe compiler to target all three technologies without having to learn a new syntax. Also, as haXe, Flash, JavaScript and Neko can function on Windows, Mac OS and Linux, many developers will be able to embrace haXe without having to change their preferred operating system.

As time goes on, further bridges to other technologies are being developed. With an emphasis on Neko, the haXe language can be further enhanced without having to modify the compiler in any way. This has led to other exciting developments such as the SWHX (ScreenWeaver for haXe) framework for creating functional desktop applications using a Flash GUI (Graphical User Interface) layer or our own Neko Media Engine (NME), which wraps the Simple DirectMedia Layer (SDL) framework and functions as a great 2D games API.

The point here is that the haXe compiler is not a new technology, merely a program that converts a new language to numerous existing technologies.

The benefits to using haXe are astronomical, but can be summarized by the following points:

- Use of existing technologies are stretched to their fullest capabilities.
- Boundaries between developers are lowered and more ground is provided for them to collaborate.
- Knowledge of the development team is increased as they write code that extends to the technologies of their peers.
- Projects are built rapidly, with less errors and platform concerns.
- Projects become easier to maintain, as all team members are able to understand the language syntax regardless of deployment technology.
- The haXe classes developed for one technology can later be used to compile for other technologies.

Compiling to Flash

Flash is a fantastic platform. So fantastic in fact, that no other tool has been able to knock it out of its position as the most popular multimedia platform for the Web, and it will take a lot of effort to find a viable contender. With its ability to run on numerous machines and operating systems, it is safe to say that Flash will be around for some time to come, despite rumors that Microsoft’s new developments will be a threat to Flash.
Part I: The Core Language

The Flash IDE, synonymous to developers who create Flash movies, contains its own version of a Flash file compiler, as does the new Flex Builder application from Adobe. These tools are probably the forerunners in commercial Flash development, but are not needed to create complete Flash applications.

Since the introduction of ActionScript 2.0 — the scripting language behind Flash movies — developers have had the ability to write applications using pure code. Unfortunately, though, the Flash IDE is not the most ideal environment for building Flash files in this way, as it is aimed primarily at designers. Flex provides a better solution, but still has drawbacks.

If you contemplate both Flash and Flex and their relative language syntax — ActionScript 1 and 2 for the Flash IDE and MXML & ActionScript 3 for Flex Builder — you will see two very different programs and languages compile to the same platform. Granted, the Flash IDE compiles to Flash versions 9 and below (ActionScript 1, 2, and 3) whereas the Flex Builder IDE compiles to Flash version 9 alone (ActionScript 3), but they inherently perform the same feat. haXe is able to perform the same routine of compiling from a source language syntax to the Flash byte code specification, much like the Flash IDE and Flex Builder, except that haXe is able to compile a single language to both the complete Flash 6 to 8 specifications and the Flash 9 specification. That’s pretty impressive in our book (pun intended).

haXe makes all of this possible by providing a language that is loosely coupled to the output, which is why it is able to support so many platforms. All haXe needs to understand is which equivalent structures for each platform map to the haXe structures and how it should be serialized in the output file.

The Flash SWF file, which is one such output, is a document containing a series of codes in the form of bytes. Each byte represents media, functions, variables, or properties that make up the content of the movie and describe to the Flash virtual machine (or player) the exact content and functionality of the movie. When compiling to Flash, haXe produces the same SWF output and provides all of the same features as the official Flash compilers produced by Adobe, though of course, certain functions within the haXe library may not be supported depending on which version of Flash you are compiling against.

You can handle this using compiler directives that allow different portions of code to be compiled depending on the target technology.

For those of you who are used to the MTASC compiler by Nicolas Cannasse for ActionScript versions 8 and below, haXe steps in as the successor, reducing further development of the MTASC compiler to bug fixes only.

When MTASC (Motion-Twin ActionScript Compiler) was released several years ago, many developers saw, for the first time, their first break into the world of Flash. Previously, many developers would complain that the Flash IDE was far too geared toward the designer and left little leverage for the developer. MTASC changed all of that by offering an all-code entry point using tools they were already familiar with.

haXe follows this developer-friendly route, though with its powerful yet friendly syntax, it also offers designers the chance to tinker in the world of the developer.

Compiling to JavaScript

JavaScript has been around for some time, but for client-side browser scripting, there is no competition. While Internet Explorer provides access to the Visual Basic scripting interpreter, JavaScript is still the only scripting language supported by the majority of browsers, and so is the only choice for thousands of developers worldwide.
Chapter 1: Introducing haXe

Each of the well-known browsers supports quite a variation of the JavaScript API. The most noted differences are those between the Internet Explorer JavaScript engine and the Mozilla JavaScript engine, which have been the source of much pulling of hair for web developers everywhere for a number of years. When you build applications that rely heavily on client-side scripting over various browser types, it is a necessity to include numerous hacks and tricks to avoid facilitating functionality that performs well on some browsers, yet poor on others. When compiling to JavaScript with the haXe compiler, haXe provides a set of functions that form a small framework for maintaining suitable cross-browser JavaScript functionality.

Compiling to Neko

haXe compiles to the Neko byte code format, in a similar way to how it handles Flash byte code. The compiled files are used by the Neko virtual machine in a very similar way to the Flash virtual machine, though any player support must come from a third-party library.

Usually, one would write for the Neko virtual machine for the purpose of creating server-side web logic using the mod_neko module or to create a desktop application for use in a command console or batch file. Using third-party modules, it is also possible to create desktop applications and full network ready server frameworks of any scale.

Although not much has been mentioned yet about the Neko language, compiler, and virtual machine, they do form a fairly substantial part of this book. The Neko framework is discussed in detail in Chapter 9, “Building Websites with haXe.”

The haXe Language

New languages pop their heads out of the woodwork at a staggering rate. It seems, every time we browse the Web, we find a new language that claims some fantastic new capability or style. The haXe language, however, was designed to unite existing technologies under a single language, so the syntax of the haXe language was its most scrutinized characteristic.

haXe is, by definition, a high-level language. The primary benefits for using haXe are its simplicity and consistency. Where most languages force a user to program in a certain fashion, haXe provides a hybrid nature of many features, but at the same time, haXe strives to produce the best of all features. For example, haXe is a statically typed language, so it is important that data containers within a haXe program maintain a set data type, thus maintaining security and good coding practices. However, in keeping with the advantages of dynamic languages, you need not specify what type of data a container represents at design-time. Instead, haXe uses C++-style templates and type inference, so that you can benefit from the flexibility of dynamic types.

Another benefit of haXe is that it supports both functional programming and object-oriented programming principles, while still maintaining solid best programming practices. On the functional programming side, haXe supports type inference as already mentioned, as well as nested functions and recursion. The object-oriented capabilities of haXe, however, allow for classes, interfaces, enumerators, and getters and setters.

Throughout the course of this book, the main focus will be that of object-oriented programming techniques in haXe, though the functional programming features will be outlined for reference.
Part I: The Core Language

**The haXe Libraries**

The haXe language comes complete with various libraries that one would expect from any mature language. As standard, haXe provides support for XML, regular expressions, sockets support, and database connectivity. Unlike many languages, the haXe core library also provides a template system, a persistent database objects framework, and a remoting framework that allows for scripted communication between Flash, JavaScript, and Neko.

If a particular feature is not present in the current haXe distribution, you will be happy to know that the active community is releasing new modules all the time. Many new and exciting modules are in development that seem destined to far exceed any other language in terms of feature, speed, and ease of use. While creating your own libraries is so simple, you’ll find yourself building all sorts of creations for the sheer fun of it.

All of the core libraries are covered later in the book.

**How Does haXe Work?**

With the exception of any Neko modules you may be using in your project, all haXe files are merely text documents with the extension .hx. Each file contains classes that make up your application, regardless of what platform you will be exporting to.

Your application will also make use of a number of standard classes in the form of packages, which are also in .hx files and provide reusable functionality for use in everyday applications. In all likelihood, you will probably create your own packages in order to speed up applications development as you gain experience with haXe.

As you write your applications, you will likely write code that is specific to certain platforms and some code that is relevant to all platforms. Later in the book, you’ll learn how you can separate this code for each platform so that only relevant code will compile. This helps maintain your code in a way that reduces the need to duplicate a lot of application logic.

Once you are happy with your code, you simply compile the .hx files to the requested platform, which will then produce a file readable by the target platform’s interpreter or player. This will be an SWF file for Flash players, a JS file for JavaScript interpreters, and an N file for the Neko virtual machine. Compiling is explained in Chapter 2, “Installing and Using haXe and Neko.”

**So What Is Neko?**

If haXe is the aesthetic syntactical sugar coating of programming, Neko is the almighty creator of functionality. Since discovering and falling in love with the haXe language and compiler, it didn’t take too long to discover the true power of Neko and realize that, although haXe is a breakthrough scripting language, Neko is by far Nicolas Cannasse’s greatest creation.

Unlike haXe, the standard Neko language is dynamically typed. However, also unlike haXe, the language was created to be more easily interfaced by language generators rather than directly by a programmer. The reason for this is that the Neko compiler was never truly meant to exist by itself, but to provide a powerful tool and virtual machine for existing languages. This is why haXe and Neko are considered a package or toolkit.
Chapter 1: Introducing haXe

The scripting capabilities of Neko come in three flavors:

- The Standard Neko language
- NXML
- NekoML

The standard Neko script is a procedural language definition, which you use later in this book while you prototype your own custom modules. The Neko language makes testing new modules very quick and easy, as no object-oriented structure or methods are required.

The other two Neko languages include an XML style language called NXML and a functional language called NekoML. NXML, like the procedural Neko language, is aimed at language compilers and generators. The reason for this choice is that, although standard languages are more easily read by people, XML trees are far easier for machines to generate and navigate. NXML also provides a more simple way to include file and line information for the compiled .hx files so that more accurate debugging information can be provided to the developer. By adopting NXML, any language that supports XML is able to build scripts that can then be compiled by the Neko compiler for execution by the virtual machine.

NekoML, on the other hand, is a different kettle of fish as it is styled around functional languages from the ML family, with a specific similarity to Objective Caml. ML languages provide a very organic method of processing data using nested functions and recursion that reduces the amount of used resources and is very well suited to symbolic processing and pattern matching as used by compilers. This makes NekoML the perfect language for creating your own compilers.

Although the Neko compiler was originally written in Objective Caml, it is now written in NekoML and compiled by the Neko compiler. This round-robin technique is known as bootstrapping.

As time goes on, other languages will be written for compilation against the Neko virtual machine. At the time of writing this book, there is a small group of individuals working toward building a compiler to allow Ruby scripts to run under Neko. If, after reading this book, you find yourself wanting to create a compiler for your own favourite language, drop an e-mail to Nicolas Cannasse on the haXe mailing list and he’ll steer you in the right direction.

**haXe and Neko Requirements**

While using numerous other scripting languages in the past, we have often had issues with complex installations, numerous dependent software installations, complex file structures, and all sorts of yoga-like key combinations that practically dislocate your fingers, just to get the things running. We have since given up these annoyances and left any references to the game of Twister to Lee’s six-year-old daughter.

haXe and Neko are different. They are very minimal programming platforms. All you need is the haXe and/or Neko distributions from [www.haxe.org](http://www.haxe.org) and [www.nekovm.org](http://www.nekovm.org) respectively and a computer that has either Windows 95 or above, a flavor of Linux, or Mac OSX as the running operating system.
Part I: The Core Language

**Summary**

In this chapter, you looked at:

- The issues with developing for the Internet, today
- What haXe and Neko are
- Why haXe and Neko are beneficial to developers
- What requirements are needed to run haXe and Neko

You install your copies of haXe and Neko in the next chapter.
Installing and Using haXe and Neko

Starting out with a new language can seem intimidating if you’re not sure how it relates to existing languages that you’re already comfortable with. That’s why you’ll take the traditional route. In this chapter, you’ll learn the very basics that will help you on your way to mastering haXe. You’ll learn the following:

- How to manually install haXe and Neko on Windows and Linux machines
- How to use the haXe automatic installer
- How to write a simple application in haXe
- How to compile your program to the Flash SWF format
- How to compile your program to the JavaScript format
- How to compile your program to the Neko format for the Web and desktop
- How a basic haXe program is structured
- How to build and execute an .hxml compiler file

Installing haXe

haXe is an incredibly easy-to-install programming tool, thanks to the automatic installer packages, which enable a simple deployment of both haXe and Neko for any of the supported operating systems. The simplicity with which haXe can be installed and used is one of the finer points of the language that many scripting languages cannot lay claim to.
Using the haXe Automatic Installer

To install haXe and Neko on your machine, regardless of what operating system you are using, first download the appropriate automatic installer package from www.haxe.org/download. Once downloaded, run the application. This presents you with a window containing a text field, as detailed in Figure 2-1, which displays the current status of the installation.

![haXe Installer](image)

Figure 2-1

You are then be asked if you’d like to install haXe and Neko to your machine. Click Yes to allow the installer to download the latest binaries of haXe and Neko for your particular operating system and to unpack these files to the necessary location. On Microsoft Windows, this is the directory C:/Program Files/Motion-Twin/haXe and C:/Program Files/Motion-Twin/neko, whereas on Linux, the application files for both packages will be split among the various system directories, which usually include the directories /usr/lib and /usr/bin.

Manually Installing haXe on Windows

To manually install haXe on Windows, first go to the haXe website at www.haxe.org/download and acquire the latest manual install package. If you like, you could be more adventurous and download the latest source of haXe from CVS, though explaining how to do this on Windows is beyond the scope of this book. However, there are plenty of references on the web that will guide you through doing this.

Once you have your haXe distribution zip file safely downloaded to your machine, extract the files to a directory somewhere on your hard drive. I tend to extract mine to C:/haXe, as it provides a nice short path and saves having to click through endless layers of directories to find the program executables.
Within the newly extracted files, locate the file haxesetup.exe and double-click it. You should then be presented with an information dialog, as shown in Figure 2-2:

![Figure 2-2](image)

And that’s it! You can now start creating your haXe applications. The haxesetup.exe file created a system environment variable so that you can call the haXe executables no matter where you are within a command window. No other steps are needed to make use of haXe, so if you’re itching to get started, you can now skip to the first example on how to build and compile a haXe application.

**Manually Installing haXe on Linux**

Unlike the Windows installation, Linux requires a slightly different route. As with most Linux applications, installing haXe requires that you first download the latest source from a CVS repository. Once this is done, you then compile the sources with Objective Caml. Most distributions of Linux already come with versions of CVS and Objective Caml that you can use, though you can check to see if either already exists on your system by entering the following:

```
> cvs --help
> ocaml --help
```

If you receive text stating that the command `cvs` or `ocaml` is unknown between each line, then have a look on your Linux distribution CD for an installable package of these two programs. Objective Caml can also be downloaded from the official website at [http://caml.inria.fr/ocaml/](http://caml.inria.fr/ocaml/).

Once you are sure CVS and Objective Caml are installed on your system, open a terminal window and navigate to a directory where you would like the haXe source files downloaded. Next, enter the following:

```
> mkdir haxe
> cd haxe
```

This creates a new directory called `haxe` and then proceeds to set the new directory as the current working directory. You can now download and compile the source by entering these lines:

```
> wget http://haxe.org/_media/install.ml
> ocaml install.ml
```
Part I: The Core Language

In the preceding code, the `wget` command retrieves the `install.ml` script from the haXe website and executes it. The `install.ml` is an Objective Caml script that automates the download and compilation of the haXe source files. You shouldn’t need to do anything else to compile the sources. Though, you do still need to put your files into a suitable directory on your PC so that they can be accessed readily and set up any necessary system variables so your PC knows where to find the files. To do this, enter the following lines at the command prompt:

```
> sudo mkdir /usr/local/haxe
> sudo mkdir /usr/local/haxe/bin
> sudo cp haxe/bin/* /usr/local/haxe/bin
> sudo cp -r haxe/std /usr/local/haxe
```

Here, the first two lines create some new directories in a standard location; the command `sudo` ensures that it is done using the root user credentials. Next, the newly compiled haXe files are copied to the new directories. Now, continue by entering these lines:

```
> sudo ln -s /usr/local/haxe/bin/haxe /usr/local/bin/haxe
> sudo grep "HAXE_LIBRARY_PATH" /etc/environment || echo "export HAXE_LIBRARY_PATH=/usr/local/haxe/std:" >> /etc/environment
```

In the first of the two lines, a mapping was created for the haXe executable in the `/usr/local/bin` directory and the second line set an environment variable so that the haXe executable can be accessed from anywhere on your PC.

Having successfully achieved these instructions, you should now be able to start creating your very own haXe applications, though you might want to install Neko before you get your hands dirty.

Installing Neko

You should read this section if you want to manually install Neko on your operating system. If you don’t want to do this manually, read the previous section on how to automatically install haXe, which includes the Neko install.

When you manually install haXe for Windows, a copy of Neko is also installed allowing you to run your Neko files. However, installing the Neko distribution from the Neko website allows for some extra development capabilities. These include being able to compile your own Neko modules in C for use in haXe and being able to examine some of the source files included with the distribution so that you can get a good look at how the Neko language can be used. Also, Linux users compiling from source will not have the luxury of having Neko installed for them, so they must follow along, too.

Manually Installing Neko on Windows

Installing Neko on a Windows operating system requires a slightly different route over the haXe install as an environment variable needs to be manually set so that Windows can locate the Neko files when you want to compile your Neko applications. If you have already installed the haXe distribution, you can safely skip setting the environment variable, as a suitable variable will have already been set.
Chapter 2: Installing and Using haXe and Neko

Neko has its own website as it is a powerful tool in its own right. To download the latest build of the Neko platform, navigate to the download section at www.nekovm.org and locate the Neko download for the Windows distribution. Once you have downloaded the zip file to your PC, extract the files to a directory on your hard drive. I tend to use C:\Neko for the same reasons as my choice of haXe directory.

Now you need to set the environment variable, so if you haven’t installed haXe yet, or you don’t plan to use haXe, then read on; otherwise you can safely skip the next part and start getting your hands dirty.

To set the environment variable, you need to call up the System window from the Control Panel directory. Once open, click the Advanced tab. At the bottom of this panel, click the Environment Variables button to display another window, which is divided horizontally into two sections, as detailed in Figure 2-3. Under the System Variables section, click the Add button.

A new dialog window should appear with two labeled text fields. Enter NEKOPATH as the variable name and the location of the Neko directory as the variable value, as shown in Figure 2-4. Now, click OK. The environment variable should be effective immediately, so Neko should now be installed and accessible from the command line. You can test this by typing neko at the prompt and pressing Return.
Part I: The Core Language

![New System Variable](c02.indd)

Figure 2-4

**Installing Neko on Linux**

For Linux users, installing Neko should be almost identical to installing haXe, only this time, you don’t have an install script to automate the compilation process.

To begin, you need to download the Neko files from CVS:

```bash
> cvs -d:pserver:anonymous@cvs.motion-twin.com:/cvsroot login
> cvs -d:pserver:anonymous@cvs.motion-twin.com:/cvsroot co neko
```

If you typed everything correctly, this should have created a temporary neko directory and filled it with source files.

*Compiling the Neko source files may require that you have certain packages already installed on your system. These include zlib, SQLite3, and MySQL libraries. Chances are, you already have the ubiquitous zlib library, while the other two are required only if you want to access Sqlite or MySql databases through Neko.*

Now, enter the following lines:

```bash
> cd neko
> make
```

You should be presented with lists of text describing the compilation procedure. You may be prompted to specify the location of an installed library if the compiler has difficulty locating it automatically. Also, you can skip the compilation of the SQLite and MySQL extensions by entering `s` when prompted. Once finished, you then have the job of transferring these newly compiled files to a suitable place on your machine. You do this in the same way as you did with the haXe installation:

```bash
> sudo mkdir /usr/local/neko
> sudo cp bin/* /usr/local/neko
> sudo ln -s /usr/local/neko/neko* /usr/local/bin/
> sudo ln -s /usr/local/neko/libneko.so /usr/local/lib
> sudo grep "NEKOPATH" /etc/environment || echo "export NEKOPATH=/usr/local/neko" >> /etc/environment
```
Chapter 2: Installing and Using haXe and Neko

As before, these lines create new system directories for your files and then copy and link them. When all are where they should be, an environment variable is created so that the system can locate the files at will.

You are now ready to develop with Neko!

Hello World! in Triplicate

So, now you get to actually compile something in haXe, albeit something simple. Almost every programming book available starts the reader off with a “Hello World!” example, whereby the resulting application will display a nice friendly message. You’ll follow this tradition. The exception, however, is that you will be compiling this example to run on three different platforms.

Here is the code you are going to compile. It is not important at this time that you understand what the code does, as the language will be discussed throughout the first part of the book.

```haxe
class HelloWorld
{
    public static function main()
    {
        trace("Hello World");
    }
}
```

You’ll need a text editor to create the file. You can use Notepad on Windows or vi on Linux, but be sure that the text editor saves to a plain text format and doesn’t add any extra text style tags or you’ll find your code will not compile.

At the time of writing this book, plug-ins for numerous IDE’s exist that should make development easier, but for the purposes of having a complete outline of creating and compiling an application, stick with a plain text editor.

Once your editor is open, enter the previous code and save the file as HelloWorld.hx in a new directory. Make sure you enter the code exactly as it is listed previously. haXe is case sensitive, so function and Function would be read differently by the compiler.

Compiling Your First haXe Application

Compiling an application with haXe is similar to compiling an application with Java and many other command-line compilers. You simply enter the compiler command followed by the required arguments at the command line. The issue with this procedure, however, is that many applications you want to compile will require a large number of arguments that are too long to reasonably enter at the command line. For this reason, haXe provides a batch file style compilation whereby all of the required arguments are entered into a text file with the extension .hxml. You can then run this file against the haXe compiler to compile your application.
Try this now with the Hello World example. Create a new text file with the following content and save it to the application directory with the filename `HelloWorld.hxml`:

```
-swf helloworld.swf
-main HelloWorld
```

Note how this content is split between two lines. For each parameter you add to the `.hxml` file, you need to separate them with line breaks. This then allows the compiler to read the content as separate commands. As you can probably guess from the parameters you’ve listed, your `.hxml` file compiles to the Flash SWF format. You can run this compile script by navigating to the working directory at the command prompt and entering the following:

```
cmd> haxe HelloWorld.hxml
```

*Windows users can also compile their applications by simply double-clicking the `.hxml` file. This facility is not available on Linux and Mac OS.*

Once you have created and run your compile script correctly, a new file is created in your application directory with the name `helloworld.swf`. If not, an error stack is displayed in the command window, which lists all possible errors found by the compiler. Often, fixing the first error in the list is enough to justify all the remaining errors in the stack and allow for a successful second attempt. You should always work through the error stack from the top down. Each time you fix an error in your code, recompiling the script may drastically reduce the list of errors being displayed.

Upon successfully compiling your script, you should be able to run the SWF file with a Flash player. If you don’t have one installed, try running the file with your favorite web browser as the browser’s Flash plug-in can often run an SWF file without the need to create an HTML container. The Flash plug-in can be acquired from the Adobe website at [www.adobe.com](http://www.adobe.com).

When you finally display the Flash movie, it should present the text:

```
HelloWorld.hx:5: Hello World!
```

Congratulations! You have just compiled and run your first haXe application.

**How Your Program Was Compiled**

So, what have you done here? In the case of the previous example, the compiler read the information provided by the `.hxml` file, which told the compiler to load and lexically scan the `HelloWorld` class. This is the base class. When you write classes in haXe, it is important to name the files after the classes they contain. This way, when the compiler processes your code, it knows what files to load based on the names of the classes. As the compiler scanned the base class, it checked for any new class names, which it also loaded and lexically scanned. Once all of the haXe code was loaded and scanned, it proceeded to convert the syntax to the equivalent SWF byte code. The compiler then constructed the necessary SWF byte codes required in all Flash files and combined the two into a single file.

The same process is performed with any of the supported platforms, the only difference being that the compiler translates the resulting code differently. For example, when compiling to JavaScript, the compiler will scan the code and form the same structures in memory as before. However, when
constructing the resulting JavaScript output, the compiler produces a small framework of cross-browser functionality before continuing to add all of the translated functionality.

**HXML Compiler Files**

Now take a closer look at the .hxml file. The content of the .hxml file you created to compile your HelloWorld application contains switches that are notable by the preceding hyphen. These switches act as commands that tell the compiler how you want the compiled output to be constructed. For example, the first line of your file displays `-swf helloworld.swf`. This tells the compiler that you want to create a file called `helloworld.swf` that is of the SWF type. The next line displays `-main HelloWorld`. This line tells the compiler that your entry method is found in the HelloWorld class, which is contained in a file called `HelloWorld.hx`. Each switch you add to your .hxml file should be separated with a new line.

The `-main` switch is important, as it allows haXe to set the starting method that will be called when the resulting movie or script is run. If you fail to specify a `-main` switch, then the resulting output will be created as a non-executable library. You’ll look at this more later in the chapter. The value you set for the `-main` switch should be a class name that contains a static method called `main`. This method is where you would normally enter all of your initialization code for the entire haXe application.

When compiling your code, other switches can be used that further affect the output created by the compiler. For example, to change the dimensions of your SWF movie, you could add the following:

```
-swf-header 400:300:30:0000FF
```

This would set your movie to 400 pixels wide by 300 pixels high with a speed of 30 frames per second and a blue background.

You can find a full list of switches for each of the three supported platforms at the end of this chapter. Though, to help aid you with compiling the examples, .hxml file content is provided throughout the book where necessary.

**Compiling to Neko**

To compile your code to the Neko format, open the `HelloWorld.hxml` file and change the code so that it looks like the following:

```
-neko helloworld.n
-swf-header 400:300:30:0000FF

-main HelloWorld
```

As you can see, the only changes you made to your .hxml file is to change the original `-swf` switch to a `-neko` switch and to change the extension of your resulting file from `.swf` to `.n`.

If you compile this now in the same way as the previous example, the haXe compiler should create a file called `helloworld.n` in your application source directory. If not, then make sure that no other switches exist in the .hxml file other than the two lines listed previously. Many of the switches you would use to compile to one format will not be supported by another format, so you must be sure that you supply the correct data for a successful compile.
Part I: The Core Language

Running the Neko Example

Neko is a very flexible platform, and as such, there are several ways to run a Neko application. The first way to run the example would be to enter the `neko` command, followed by the path to the file to be executed. At the command line, navigate to the working directory and enter:

`> neko helloworld.n`

If the file executes successfully, then the command window should display the same string as the Flash example.

If you like, you can run all of your Neko applications in this way, though this method does provide a few drawbacks. For example, in order to run the application, you will need the aid of an open command window or batch file to handle the `neko` command execution. Also, calling the `neko` command requires that an environment variable is set to provide the operating system with a way of finding the Neko executable. This will certainly not be a good idea if you plan to distribute your application.

Luckily, Neko comes with a utility toolkit that can convert your Neko application to an executable file, so your users will be able to execute your application by double-clicking its icon or by entering the executables filename into the command line.

To do this, you need to enter the following at the command line:

`> nekotools boot helloworld.n`

If successful, you should now have a file called `helloworld.exe`, or simply `helloworld` on Mac OSX and Linux, in your application directory.

Running as a Web Application

Running a Neko application on the desktop is one way to execute your code, however, the primary purpose of the haXe and Neko partnership is to provide a toolkit for web developers. While desktop applications development is a plus, you will definitely want to use Neko to serve your haXe applications to a browser.

To deploy a website with Neko to a server, you need to install the `mod_neko.ndll` or `mod_neko2.ndll` modules within an Apache web server. You learn how to do this later in the book, but for now, you will make use of another feature of the NekoTools utility—the NekoTools web server.

The NekoTools web server is a great tool for building and testing your web-based haXe applications. While not powerful enough to use as a production web server, it certainly provides all the features you need to test your Neko applications with minimal fuss. The NekoTools web server will save you hours of fiddling with transferring files or stopping and starting an Apache install. The server requires no prior setup, except that you need to have haXe and Neko installed correctly on your development machine.

To start the NekoTools web server, go to the command line and enter the following:

`> nekotools server -p <port> -h <hostname> -d <directory>`
Here, `<port>` should be a free port number on your machine, `<hostname>` is the domain of the test website, and `<directory>` is the location of your Neko application. The default for the port number is 2000 and the hostname is localhost, which should be sufficient in most situations. Also, you should not need to specify the directory switch if your current directory is the application directory. This means, in most situations, you could probably get away with starting the server by navigating to your application directory and entering the following at the command line:

```
   cmd> nekotools server
```

Now, open a browser and navigate to `http://localhost:2000/helloworld.n` or other URL if you have supplied different values to the defaults. If you were successful in starting the NekoTools web server, you should see the usual Hello World! greeting, much like the previous examples. If, however, you do not see this result or your web browser complains that it cannot find the specified web page, it could be that you have not set the correct directory when calling the URL. Don’t worry if this is the case, as you can modify the root location of the NekoTools web server at any time. Simply go to `http://localhost:2000/server:config` and you will be presented with the option to change the directory location. This is great if you are testing multiple applications intermittently.

**Compiling to JavaScript**

JavaScript is quite an ambiguous language as, like most scripting languages, it can be applied to various environments. For example, one could find a JavaScript engine used as a server-side tool or embedded within a large desktop application. However, when describing JavaScript with haXe, you are specifically targeting the JavaScript engine used by most Internet browsers.

Although the JavaScript specification used by web browsers can often vary from browser to browser, the haXe platform provides a number of functions within its compiled output that aid in providing a bridge between the different browsers, so that code run on one browser should function similarly when run on another browser. This is an added benefit of separating the language from the target environment, as haXe does.

Like in the previous examples, you can compile the hello world example for JavaScript by making a simple alteration to the original `.hxml` file. Open the file now and replace its content with the following code:

```
   -js helloworld.js
   -main HelloWorld
```

As before, all you have to do is to change the first switch to a `-js` switch and then change the file extension of the output file to `.js`.

Compiling should now produce a file called `helloworld.js` as expected. However, you will not be able to run this file, yet. First, you need to create an HTML file that will call your JavaScript code.

Unlike Flash, JavaScript is heavily associated with HTML. The JavaScript DOM (Document Object Model) provides a hierarchy that maps to the structure of an HTML document and exposes hooks to each of the HTML elements. As a result of this, it is necessary for an HTML page to call its associated JavaScript document rather than allowing JavaScript code to run on its own.
Part I: The Core Language

Create a new document called helloworld.htm, open it with your text editor, and enter the following code:

```html
<html>
<head>
  <title>Hello World</title>
</head>
<body>
  <span id="haxe:trace"></span>
  <script type="text/javascript" src="helloworld.js"></script>
</body>
</html>
```

Now save the file in the application directory and open it in a web browser. You should now see the usual output in much the same way as when you called the Neko file with a browser.

Although the Neko and JavaScript examples look similar, they are actually quite different. In the Neko example, the Hello World! string was displayed statically in the browser, whereas in the new JavaScript example, the `span` tag in your HTML file acted as a container that provided the JavaScript code with a point in which to enter the Hello World! string.

To prove this point, reopen the HTML file and place the `span` tag after the line that loads the `helloworld.js` file. Now save and run the HTML file. If you did this correctly, the browser should now be empty and the dialog shown in Figure 2-5 is displayed. This is because, although the JavaScript file was executed, the `span` tag was not yet available to the JavaScript DOM and so the code had nowhere to place the output.

![Figure 2-5](image)

Creating HTML documents for your haXe JavaScript is discussed later in the book.

Program Structure

Now, examine how your program was written in haXe. When writing an application using the haXe language, you will extend upon the class hierarchies that already exist in the haXe framework. These classes will eventually form one of two purposes: a runnable program or a code library. Both hierarchical
structures are identical in approach, except that a runnable program will expose a static function called `main` that will be run when the program is first executed, just as you have in the HelloWorld example:

```haxe
public static function main()
{
    ....
}
```

This is known as the entry point and is responsible for initializing your application and calling the logic that will facilitate the rest of your program. If the main function is missing from your application hierarchy, your code cannot be compiled as a runnable program, but will still be accessible by other applications that load the compiled files as a module.

The main function can exist among your larger application framework or separately in a class of its own. Very often, you will create a pilot class that will function as the primary messaging center of your application and will include a static main function for initialization. Other times, however, you may prefer to abstract this function. Regardless of which route you choose, a runnable application will always require a starting class, even if that class will not be instantiated.

Class instantiation is part of object-oriented programming methodologies. While you can use haXe without constructing all of your code as objects, you will still be required to structure your code within class blocks:

```haxe
class MyClass
{
    ....
}
```

Even a program contained in a single file will need at least one class block to contain the various functions that make up the application. The primary class accessed when the program is first run will not yet be instantiated, which is why the main function that serves as the application entry point is tagged as static.

You learn more about using classes later in the book.

**Compiler Switches**

Compiler switches are important for setting the environment of your applications that aren’t possible within your application code. Normally, you will only specify the output filename for your application and the name of the class containing the static method main used to initialize the rest of your application. However, in the case of Flash movies, you can also specify the dimensions, target player version, and any external SWF files you want to embed in your output movie.
### Part I: The Core Language

#### Switch Parameters Description

<table>
<thead>
<tr>
<th>Switch</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flash</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-swf</td>
<td><code>&lt;filename&gt;</code></td>
<td>Specifies the filename of the output file.</td>
</tr>
<tr>
<td>-swf-lib</td>
<td><code>&lt;filename&gt;</code></td>
<td>Specifies the location and filename of the existing swf file to embed.</td>
</tr>
<tr>
<td>-swf-header</td>
<td><code>&lt;width&gt;:&lt;height&gt;:&lt;fps&gt;:&lt;hex bg&gt;</code></td>
<td>Sets the dimensions, speed, and background color of the Flash file.</td>
</tr>
<tr>
<td>-swf-version</td>
<td><code>[6 – 9]</code></td>
<td>Specifies the Flash Player version to export.</td>
</tr>
<tr>
<td>--flash-strict</td>
<td><code>&lt;none&gt;</code></td>
<td>Compiles using a more strict Flash API.</td>
</tr>
<tr>
<td>--flash-use-stage</td>
<td><code>&lt;none&gt;</code></td>
<td>Places object found on the stage of an included SWF library onto the output files stage.</td>
</tr>
<tr>
<td><strong>Neko</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-neko</td>
<td><code>&lt;filename&gt;</code></td>
<td>Specifies the filename of the output file.</td>
</tr>
<tr>
<td><strong>JavaScript</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-js</td>
<td><code>&lt;filename&gt;</code></td>
<td>Specifies the filename of the output file.</td>
</tr>
<tr>
<td><strong>All Platforms</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-main</td>
<td><code>&lt;class name&gt;</code></td>
<td>Specifies the class that contains the static method main.</td>
</tr>
<tr>
<td>-cp</td>
<td><code>&lt;path&gt;</code></td>
<td>Adds a directory to scan for source files.</td>
</tr>
<tr>
<td>-xml</td>
<td><code>&lt;filename&gt;</code></td>
<td>Generates an XML file with the types information for the specified output format.</td>
</tr>
<tr>
<td>-lib</td>
<td><code>&lt;library[:version]&gt;</code></td>
<td>Uses a haXelib library (discussed later in the book).</td>
</tr>
<tr>
<td>-D</td>
<td><code>&lt;variable&gt;</code></td>
<td>Defines a conditional compilation flag.</td>
</tr>
<tr>
<td>-resource</td>
<td><code>&lt;filename&gt;</code></td>
<td>Adds a named resource file.</td>
</tr>
<tr>
<td>-exclude</td>
<td><code>&lt;filename&gt;</code></td>
<td>Forces a file to not be compiled.</td>
</tr>
<tr>
<td>-v</td>
<td><code>&lt;none&gt;</code></td>
<td>Turns on verbose mode.</td>
</tr>
<tr>
<td>-debug</td>
<td><code>&lt;none&gt;</code></td>
<td>Adds the debug information to the compiled code.</td>
</tr>
<tr>
<td>-prompt</td>
<td><code>&lt;none&gt;</code></td>
<td>Prompts on error forcing the command window to remain open.</td>
</tr>
<tr>
<td>-cmd</td>
<td><code>&lt;command&gt;</code></td>
<td>Runs the specified command after a successful compilation.</td>
</tr>
</tbody>
</table>
Chapter 2: Installing and Using haXe and Neko

<table>
<thead>
<tr>
<th>Switch</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--override</td>
<td>&lt;none&gt;</td>
<td>Forces all overridden methods to require that they are tagged with override.</td>
</tr>
<tr>
<td>--no-traces</td>
<td>&lt;none&gt;</td>
<td>Excludes all traces from the compiled output.</td>
</tr>
<tr>
<td>--gen-hx-classes</td>
<td>&lt;filename&gt;</td>
<td>Generates hx headers from an SWF version 9 file. Useful if you want to extend an existing SWF version 9 file.</td>
</tr>
<tr>
<td>--next</td>
<td>&lt;none&gt;</td>
<td>Enables several haXe compilations in one .hxml file.</td>
</tr>
<tr>
<td>--altfmt</td>
<td>&lt;none&gt;</td>
<td>Uses an alternative error output format.</td>
</tr>
<tr>
<td>-help / --help</td>
<td>&lt;none&gt;</td>
<td>Displays this table of information.</td>
</tr>
</tbody>
</table>

Much of what is detailed here is beyond the scope of this book. However, where necessary, this book examines some of the more useful switches when they become applicable.

Summary

Now that you’ve scratched the surface, you need to delve into the inner workings of haXe and start creating usable content. So far, you covered:

- How to install haXe and Neko
- How to create an .hxml compiler file and use it to compile a haXe application
- How to modify the .hxml file to compile to each of the languages supported by haXe
- How to run the compiled output in their native environments

Now that you know how to use the haXe tools, it’s time to introduce you to the language.
Learning the Basics

haXe is a language, first and foremost. This means that it has a defined structure and a grammar consisting of keywords and contextual rules. In this chapter, you’ll examine the grammar of the haXe language and become familiarized with the various functions and data types it has to offer.

Specifically, you will look at:

- The data types supported by haXe
- Using Variables
- Type inference
- Working dynamically with untyped and Dynamic
- Casting data from one type to another
- Grouping data into Arrays, Lists, and Hash Tables
- Working with operators
- Manipulating strings
- Using the Math functions
- Using the String functions

The haXe Hierarchy

haXe is an object-oriented language. This means that the language is structured in a hierarchical format whereby simple structures of data form the basis and content of more complex data structures. You are introduced to the object-oriented features of haXe in Chapter 5, “Delving Into Object-Oriented Programming,” but for now, it is important to understand that the haXe language relies upon a hierarchy throughout its entire makeup.
Part I: The Core Language

Many of you who are used to programming in languages such as C++ and Java will appreciate the flexibility of the haXe language and may be somewhat familiar with its grammatical style. The haXe language provides an extensive range of types — classes, functions and the like, which center on a small group of keywords. These are:

- class
- function
- enum
- anonymous
- unknown
- typedef

These keywords are used to construct and represent the containers of one or more values and ultimately affect the way values are handled in different contexts.

The Standard Data Types

Standard types, or standard data types, are the ones you use in your application to store simple values. The standard types in haXe include:

- Int
- Float
- String
- Bool
- Void
- Dynamic

There are a couple of other types that can be considered part of the standard type list, but they are more of a toolset and form the basis of more complex topics discussed later in the book, so you will not find them here.

Unlike many languages, the standard types in haXe are more so containers or identifiers that embody the simple values they represent. This provides an avenue with which to leverage object-oriented programming in haXe, by facilitating a means to literally extend the values type into a more complex object.

Variables

haXe is a statically typed language. This means that, when you declare variables, the compiler needs to know what type of data will be assigned to the variables so that it can deal with them efficiently. There is some debate in the programming communities that dynamically typing languages is often a better
Chapter 3: Learning the Basics

solution. Dynamic typing is where variables can contain data of any type at any time, often to the point that variables do not even have to be declared, as long as they are assigned data before they are queried. Once a variable holds a value, that data can then be referenced as another type without the use of casting. Programmers who opt for dynamically typed languages will often comment at how much quicker they can prototype and develop their applications and often find this lack of type restriction as a respite to the more lower-level languages they were used to. However, such programming techniques lead to untidy code and a greatly increased possibility for error leading to applications that are unstable and hard to maintain. Also, applications built with such languages tend to be quite a bit slower and more memory consuming than statically typed languages as the virtual machines that execute the code have more to deal with when creating, casting, and destroying the data structures used by the language. Regardless, the debate between static and dynamic typing will likely persist, as the taste of others will always vary considerably. For now, however, it is better to acknowledge that as haXe is statically typed, it is optimal in terms of speed and size, and the tidy syntactical requirements can only help in making you a better programmer.

Needless to say, static typing requires variable declarations. Variables in haXe need to be declared before they can be used so that an appropriate structure and memory allocation can be reserved for your data. You declare variables using the following syntax:

```haxe
var varName : Type;
```

The keyword `var` is similar to Visual Basic’s `Dim` keyword, which tells the compiler that the following line will be a variable declaration. Without the `var` keyword, the compiler will expect an expression to follow such as an assignment of data into the variable, which of course will result in a compiler error, as the variable is not yet declared.

Following the `var` keyword is the name or label of the variable. You can label your variables anything you like, as long as the variables contain only letters, numbers, or the underscore (_ character. Also, you must note that although a variable name may contain numbers, it cannot start with a number.

The final part of the variable declaration is the type. The type name must begin with an uppercase letter. Assigning a type alerts the compiler to the type of data you wish to store in the variable, so that checks can be made throughout the rest of your code, thus reducing possible errors at run time.

You can initialize a variable with data when declaring it, though it is not mandatory to do so. To initialize a variable, simply follow the declaration with an assignment operator (=) followed by the value to initialize your variable, like this:

```haxe
var varName : Type = "dataToAssign";
```

The data you supply to the variable declaration can be either literal data, as in the preceding example, or data contained within an existing variable.

To declare a variable, enter the following code into a new file and save it as `VariableAssignment.hx`:

```haxe
class VariableAssignment
{
    public static function main()
    {
        (continued)
```
Part I: The Core Language

(continued)

```haxe
var flt:Float = 10.5;
var str:String = "The quick brown fox.";
trace(flt);
trace(str);
}
}
```

Now, compile the code using the following `.hxml` content:

```
-neko variableassignment.n
-main VariableAssignment
```

When you execute the code, you should be presented with the following output:

```
VariableAssignment.hx:7: 10.5
VariableAssignment.hx:8: The quick brown fox.
```

When the code is compiled, the compiler can compare the values passed to the variables with the types specified for those variables. The `trace` function provides a means to convert most data types to a string representational format, which it then writes to the console.

The problem that may arise is if you do not know the type of information that will be stored inside a variable. What if you had created a program that was to respond to data provided by an outside source, so the information you wish to store in a variable could not be decided until run time?

The answer to this is to use type inference.

**Type Inference**

*Type inference* is the process of letting the compiler guess the types at compile time, in order to provide strict typing without having to specify types for every variable. This is not the same as dynamically typing, as the data will still need to be of a consistent type. When the compiler scans your code, it attempts to ensure that the type assigned to the variable does not deviate throughout its lifetime.

*Type inference will have a great impact on your applications when you start using functions. You are introduced to functions in Chapter 4, “Controlling the Flow of Information.”*

Enter this code into a new file and save it as `TypeInference.hx`:

```haxe
class TypeInference
{
    public static function main()
    {
        var myVar = 10.5;
        trace(myVar);
    }
}
```
Now compile and run this example. You should see the content of `myVar` displayed in the command window. Now modify the code to include these new lines:

```java
class TypeInference {
    public static function main() {
        var myVar = 10.5;
        trace(myVar);
        myVar = "The quick brown fox."
        trace(myVar);
    }
}
```

When you compile this example, you should be presented with an error that states the following:

```
TypeInference.hx:7: characters 8-38 : String should be Float
```

In the first example, the compiler does not register the type of the variable until it is first assigned a value. Once the value type is determined, it then binds that type to the variable, so that only that particular data type can be assigned to that variable in the future.

In the second example, the compiler noted that the variable had first contained a `Float`. It was then assigned a `String`, which cannot be immediately converted to a `Float` type, so the compiler threw an error.

**Constant Variables**

Many languages allow you to store values into variables that will remain constant, meaning that the data contained by the variable will not, and so cannot, be modified. The purpose of providing such a facility provides literal values with a human readable definition, allows for a central location for maintaining a regularly used literal, and ensures that certain values are not changed throughout your application when you know there is no reason why they should be. haXe doesn’t support this kind of restricted access to variable data, though it can help to apply this artificially by specifying a certain variable naming convention that is discernible by other developers working on your project. Normally, there are set naming conventions for keywords and objects in haXe that force your code to be more readable, and it is expected by the haXe naming conventions for you to name your variables by starting them with a lowercase letter, or in some circumstances, an underscore character. When you deal with constant values, however, it often pays to forgo this convention and provide all uppercase letters, though you are free to adopt your own system.

**The Simple Value Types**

The standard data types can be further broken down into two groups: simple types and abstract types. The simple types, which are examined in detail now, consist of `Floats`, `Integers`, `Strings` and `Booleans`. As you can see, the simple types represent associated literal values. The abstract types consist of `Dynamic` and `Void`, and represent unknown quantities.
Part I: The Core Language

**Floating-Point Numbers**

Flots represent numbers that contain a decimal point even if they are whole numbers. This is important, as 10.0 is quite different from 10 and must be stored appropriately. If you feel that a round number will need to support fractions of a number, then it helps to plan for this in advance and choose to use a Float.

haXe provides a lot of help with choosing which type to use, thanks to type inference. When you use constant numerical values within an expression, haXe automatically converts non-floating values — or integers — to Flots where necessary. Likewise, any Flots containing a trailing 0 may be converted to an integer if the expression warrants it.

In haXe, Flots are of the double precision IEEE specification utilizing 64 bits (8 bytes). This is the standard for many languages and gives extremely high precision. This same precision is carried to all three of the platforms supported by haXe, so you can be sure that you will receive similar results regardless of your target platform. Double precision floating-point numbers also have the benefit of being able to contain very large non-floating-point numbers, or Flots with a small decimal precision.

**Integers**

Integers are represented by the Int class. Ironically, the Int class extends the Float class in haXe. This is because all integers are merely floating-point numbers with a zero precision mantissa (the right side of the decimal point), while the reverse is not true, as converting a Float to an Int requires losing the mantissa precision entirely.

Most scripting languages, including the Flash SWF specification and JavaScript, allow 32 bits for their integers, which can store values of up to $2^{31} - 1$ or $-2^{31}$. The integers are signed, which means the most significant bit, the bit on the far left, is used to determine whether the value is positive or negative. Neko, however, uses only 31 bits for its integers from a 32-bit value, which can store maximum values of $2^{30} - 1$ or $-2^{30}$. The reason for this is to increase the speed at which values are accessed.

Normally, a type within a scripting language is stored within the virtual machine using a structure, like this:

```c
struct value {
    int value_type;
    void *value_data;
};
```

Here, the value_type is used to determine the type of the value, so an integer might be 1, a float is 2, and a string is 3. The actual data of the value is then stored in the value_data pointer. When the virtual machine needs to apply processes to the value_data, it has only to check the value of value_type to determine the content of value_data. While this works really well, the problem with this approach is that creating such a structure slows things down, as the garbage collector has to deal with a more complex data type. As Ints are more often used than any other type, it seemed fitting to make use of a trick to maintain the maximum possible speed. When a value structure is used by Neko, it is referenced using a pointer that occupies 32 bits. The most significant bit of this pointer is always 0. As the Int is also only 32 bits, you can store both data types in exactly the same fashion, but determine the differences...
between them by setting the most significant bit for all Int values to 1. This way, you have all the speed of a raw integer while maintaining a uniform type differentiation system.

As Neko uses the 31st bit of its integer values as the signed bit, you should ideally provide support for this limitation across all three of the supported platforms where code may be interchangeable between each platform. Therefore, when you use Ints in haXe, always make sure the value you are representing will not exceed the 31-bit limit. If you are unsure, however, then use a Float instead, which can contain far greater values.

**Which Number Type?**

You have touched on some pretty complex concepts while describing the Int and Float types. However, if you have no idea what has just been discussed and all this talk of bits and pointers has flown way over your head, it will probably help to provide a small summary. When choosing the type to use for your numerical data, you should use an Int if your value will always be less than 1073741823, more than $-1073741824$, and will always be a whole number. Otherwise, use a Float.

**Booleans**

Booleans represent one of two states: true or false. They are often used as a return value of an expression or to store the state of an argument where there can be only two possible outcomes. For example, you might decide that a flag can be kept depicting whether or not debug information should be printed while your program is running. To do this, you could create a variable of type Bool called displayDebugInfo and initialize it as false. Then, when you want all debug information to be displayed in the console, you could simply change the value of this variable to true.

Unlike many other languages, booleans in haXe are identifiers only and do not have a numerical representation. Therefore, the following is not acceptable in haXe:

```haXe
var state : Int = 1;
var valid : Bool = state;
```

This restriction to the usage of booleans in haXe should be seen as a benefit, as this kind of cross-type assignment in a statically typed language can lead to errors that statically typing is supposed to alleviate. However, if you absolutely must be able to convert other data types to a boolean, you can make use of the Std.bool() function as described later in this chapter.

**Strings**

Strings are a useful data type, especially in web-based applications where much of the data transmitted from server to client is in a textual format. The String type in haXe can represent textual values of any size, though it can prove more efficient to include very large textual strings via an external text file or by using the StringBuf (string buffer) class.

String literals are represented as characters surrounded by double or single quotes. Both have the same result, so it is up to you which type of quote you choose to adopt. When you build your string content, it is possible to add invisible characters such as tabs and newlines. These are called escaped characters and are supplied by using the backward slash (\) followed by a letter, as shown in Table 3-1. For example,
when typing columned lists, you could be tempted to separate your columns by entering, say, four space characters between each column, like this:

```haxe
var myList:String = "id    qty    name";
```

This would be fine for a single line of text, but when you enter subsequent lines and supply four space characters between each column, your results might end up looking something like this:

```plaintext
id    qty    name
1     22     Whizz Crackers
2     1      Snozz Bangers
....
21    14     Fizz Snappers
```

As you can see, although exactly four space characters were supplied between each item, the columns didn’t line up very well. To fix this issue, the space characters can be replaced with the tab escaped character `\t`, which makes the `String` declaration look more like this:

```haxe
var myList:String = "id\tqty\tname";
```

This modification to the example aligns the resulting list so that the left-hand character for each column will be correctly aligned with each other:

```plaintext
id      qty    name
1        22     Whizz Crackers
2        1      Snozz Bangers
....
21      14     Fizz Snappers
```

This functionality isn’t guaranteed in all terminals, and indeed, some applications displaying output from your haXe program may very well use a set number of space characters in place of a `\t`, but you can see its usefulness.

Table 3-1 contains a list of the available escaped characters for the `String` data type.

### Table 3-1

<table>
<thead>
<tr>
<th>Escaped Character</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\t</code></td>
<td>Tab</td>
<td>Aligns text to the next tabbed location</td>
</tr>
<tr>
<td><code>\n</code></td>
<td>Newline</td>
<td>Ends the preceding text and starts again from a new line</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>Carriage return</td>
<td>Similar to newline</td>
</tr>
<tr>
<td><code>\</code></td>
<td>Backward slash</td>
<td>Displays a backward slash</td>
</tr>
<tr>
<td><code>&quot;</code></td>
<td>Double quotes</td>
<td>Displays double quotes in a string depicted by double quotes</td>
</tr>
<tr>
<td><code>'</code></td>
<td>Single quote</td>
<td>Displays a single quote in a string depicted by single quotes</td>
</tr>
</tbody>
</table>
Abstract Types

The abstract types, for want of a better name, are identifiers that cannot be represented in a literal format. That is not to say, however, that they cannot contain literal values.

Void and Null

The Void type is a placeholder for no type or value. Very often, a type is required within areas of your code that allow the compiler and virtual machine to understand the flow of data in your program. Using the Void type tells the compiler that no data is to be expected at a particular juncture, so no type checking is required. The usual location for such a type is as the return value of — or lack of — a function, though you will also find the Void type useful when you tackle creating extensions for Neko in C.

The null type is not a type at all, but a kind of constant that means empty of value. Any variable can be equal to null, which means that, although they are of a set type, they do not yet have a value, whereas a variable of type Void would always be null, as it can have no value. This can get a little confusing when you start using pointers to C structures that you would store in a variable of type Void, but would not otherwise perform haXe-based comparisons upon such a value. The value null itself has the type unknown.

It is important to know the difference between Void and null, so that they are used in the correct context. It can often help to consider null as an absence of a value and Void as a value that is not native to haXe or as a signifier that null should be passed in its place, though strictly speaking, null is actually a value unto itself.

The Dynamic Type

The Dynamic type is like the haXe Swiss Army Knife. You’d say it’s one of the most useful types in the language, but on consideration, what kind of program can you create without Ints and Strings? You’ll look at only one use of the Dynamic type now, as some of its more advanced uses will have to wait until later in the book.

Contrary to what you have learned throughout the beginning of this chapter, there are several ways haXe can be treated as a type-less — or dynamic — language. The Dynamic type is one of those ways. Those of you who are already proficient with ActionScript 2.0 can liken the Dynamic type to the Flash Object type, though there are a number of differences that make programming with the Dynamic type a little cleaner.

To use the Dynamic type, enter the following code into a file called DynamicTest.hx and compile it to Neko:

```haxe
class DynamicTest
{
    static public function main()
    {
        var dyn : Dynamic;
        dyn = 16;
    }
}
```
Part I: The Core Language

(continued)

```javascript
var str : String = dyn;
var int : Int = dyn;
var flt : Float = dyn;

trace( str + " " + int + " " + flt );
```

If you entered this code correctly, the output should be: 16 16 16.

As you can probably imagine, had the variable `dyn` been of the `String` or `Int` type, passing the value of `dyn` to one of the other variables would have resulted in a compiler error. As it is, even though you strongly typed your `str`, `int`, and `flt` variables, the compiler noticed that you were only assigning a value from a `Dynamic` type and so your typing was removed and the three variables were set to a `null` — or `unknown` — type, which is no type at all.

In any situation, the compiler understands that the data held in a `Dynamic` type is not known at design time, or indeed run time. However, you will not be able to force data of one type into a new form. The `Dynamic` type is merely a way to provide a container for data until you can reason what should be done to that data. If you wish to convert a data type into a different type, you will need to use casting.

> When using the `Dynamic` type, one must bear in mind that the compiler will no longer be able to identify type violations. It is important that you maintain strict type rules and only apply functions that you know are valid for the data contained within a `Dynamic` type, as failure to do so will create unpredictable results and errors that are hard to locate.

### The `unknown` Type

The `unknown` type represents type-inferred variables that have yet to be assigned a type. Once a value is assigned to a variable of type `unknown`, its type will be exchanged to match that of the passed data.

### Bypassing Static Typing with `untyped`

Sometimes you will reach a point where a large amount of untyped processing is required, and simply piling all your data into lists of `Dynamic` variables will seem a messy option. Under such circumstances, you could use an `untyped` block. Here, you simply surround the code you want to free of all type restrictions with curly braces `{ }`, and head the block with the `untyped` keyword:

```javascript
var int : Int = 0;

untyped {
    int = "Some string";
}
```

Of course, although the previous example will now compile, the data contained by the variable `int` following the `untyped` expression will vary from platform to platform. In Flash, passing a string to a variable of type `Int`, even when `untyped`, will cause the data held by the variable to display as `NaN`
Commenting Your Code

Before you start getting involved with manipulating your data, it is important to note how to comment your haXe scripts. Commenting provides a great way for you as a developer to help remember what your code is doing. Certainly, when working with one or more other developers on the same project, commenting can save time and even money, by allowing others to understand what your code is doing.

Commenting in haXe follows the same format as C++ and Java, so developers of these languages should be fully at home with the haXe conventions. There are two ways to comment in haXe, and both are useful in different situations.

Block Commenting

Block commenting is more useful when larger amounts of information are needed in your code. The block comment requires two lots of symbol groups: a forward slash and asterisk (/*) before your comment block and an asterisk followed by a forward slash (/) after the comment block. This way, the compiler will continue to ignore your text, even across new lines, until the second symbol group is read.

/* This is a comment block spanning several lines */

The most common places for a comment block is at the head of each code file as a complete description of the contained classes or at the beginning of a function as the function’s description of usage.

Line Commenting

Line commenting is useful if you want to describe what a particular line of code is doing, or you want to provide a brief instruction between lines of code. You start a line comment using two successive forward slash symbols (/):

// This is a line comment
var someVariable : Float; // This is also a line comment

Anything following a forward slash pair is ignored by the compiler until the next new line, where code can continue uninterrupted. Even other comment symbols are ignored, so following a line comment with a multiline block comment on the same line will likely cause a compiler error:

var someVar = 0; // line comment /* new block comment
this line will cause an error */
Converting Data Through Casting

So far, you’ve done quite a bit dealing with types and their restrictions, and can now see how typed values are kept separate from other types by the compiler. However, a lot of the data you will deal with will require some form of conversion so that, for example, a class of type `car` can fit into a variable of type `Vehicle`. In order to do this, you need to use casting.

In truth, casting in haXe does not literally convert the data as it would in C++, for example, but instead performs an unsafe allocation of data from a source value to a variable.

Those of you who program with C++ or Java will be very familiar with casting, which involves tagging the type you wish to return to the variable or value, like this:

```cpp
double myDbl = (double) 44;
```

Here, the literal integer 44 is converted to a floating-point number before being stored in the variable `myDbl`. The cast is valid as the data types are of a similar nature, though custom parsing routines are necessary when dealing with more complex data conversions. In haXe, however, the variable is allowed to contain whichever data you like, in essence, though the compiler will expect the variable to provide the same functionality throughout its lifetime regardless of the data it contains.

The cast Function

haXe supplies a similar method to C/C++ casting in the guise of the `cast` function. You simply pass the `cast` function a value and the type of the value you want to convert it to, and the function will return the newly cast value. Here is your C++ example rewritten for haXe, using the `cast` function:

```haXe
var myFlt : Float = cast(44, Float);
```

Now, of course, you could rely on type inference to pass the literal of 44 to the variable, but under certain circumstances, this would be the most simple and primary means for conversion. Now, the `cast` function worked great for converting an integer to a floating point value, but converting a floating point value back into an integer is not possible using `cast`. Why? Well, unfortunately, casting has quite a large limitation. The variable you pass to the `cast` function must extend the type that is also passed to the `cast` function. If you remember back to when you learned about how haXe incorporates integers and floats, it is noted how the `Int` type extends the `Float` type. This meant that, while an integer is a type of float, a float is not a type of integer, so the conversion can only flow in one direction.

To get around type casting restrictions using `cast`, you can opt for a dynamic route, so that no compiler checking is performed. To do this, you simply omit the type parameter for the `cast` function:

```haXe
var myFlt : Float = cast(44);
```

This is known as an unsafe cast, and performs the same functionality as passing the data from a `Dynamic` type variable as opposed to returning it from a call to the `cast` function, such as:

```haXe
var tmp : Dynamic = 44;
var myFlt : Float = tmp;
```
As you saw earlier, this has the affect of shoehorning the value into the variable, but it still doesn’t actually convert the value as such.

Casting an Int to a Float is a bad example, as for this pair, the haXe compiler will perform this automatically. However, short of creating your own classes, this would be hard to demonstrate otherwise. Just keep in mind that the cast function will normally be needed.

Simple Type Conversion Functions

Thankfully, haXe provides yet more functions with the specific purpose of converting simple types from one type to another; these are found in the Std (pronounced standard) class. These methods are grouped based on the starting type of the data to convert, though the naming conventions don’t help in making them particularly distinguishable.

Functions contained in classes are termed methods. This term will be used to describe functions of classes throughout the rest of this book.

Table 3-2 lists the available conversion methods.

### Table 3-2

<table>
<thead>
<tr>
<th>Std Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool( val : Dynamic ) : Bool</td>
<td>Converts any value to a boolean type. Only 0 (zero), null, and false are returned as false.</td>
</tr>
<tr>
<td>string( val : Dynamic ) : String</td>
<td>Converts any value to a string.</td>
</tr>
<tr>
<td>chr( val : Int ) : String</td>
<td>Converts ASCII code to a string.</td>
</tr>
<tr>
<td>ord( val : String ) : Int</td>
<td>Converts the first letter of a given string to its equivalent ASCII code.</td>
</tr>
<tr>
<td>int( val : Float ) : Int</td>
<td>Rounds a float down and returns as an integer.</td>
</tr>
<tr>
<td>parseFloat( val : String ) : Float</td>
<td>Converts a string to a float.</td>
</tr>
<tr>
<td>parseInt( val : String ) : Int</td>
<td>Converts a string to an integer.</td>
</tr>
</tbody>
</table>

When using any of the conversion methods listed in the previous table, you must precede the method call with the Std class name using dot notation, as each of the methods are static and belong to the Std class. Here’s a quick look at using these conversion methods.

Enter the following into a new file and save it as TypeConversion.hx:

```haxe
class TypeConversion
{
    static public function main()
    {
        var myInt : Int = 45;
```
Part I: The Core Language

(continued)

```haXe
var myFlt : Float = 0;
var myStr : String = ""
myStr = Std.string( myInt );
trace( myStr );
// Outputs: 45
myFlt = Std.parseFloat( myStr ) / 2;
trace( myFlt );
// Outputs: 22.5
myInt = Std.int( myFlt );
trace( myInt );
// Outputs: 22
}
}
```

Having set up the initial variable values, the integer value 45 stored in `myInt` was converted to type `String` and passed to the variable `myStr`. Next, the value of `myStr` was converted to the `Float` type, which would have resulted in 45.0. However, to make the conversion more evident, it was divided by 2, resulting in the value 22.5. The last conversion was to then convert the floating point value back to an integer, which lost the trailing .5 leaving 22.

**Comparing the Type of a Value**

Very often, particularly when dealing with values stored in a Dynamic variable, you will need to know the type of a variable’s value before you can act upon it. The `Std` class contains a method called `is` that allows you to perform such a check. You use it in a similar way to the `cast` function, only instead of returning a newly cast value, it returns a boolean; `true` if the value is of the given type or `false` if it isn’t. Here is an example:

```haXe
var isInt : Bool = Std.is( myVar, Int );
```

This statement would return `true` if `myVar` is an `Int`, or `false` if it’s not. If the value of the variable could be one of multiple types, then you would have to repeat the process for each type that the variable might be.

**Using Arrays in haXe**

Like most languages, haXe supports grouping lists of data into Arrays and provides functionality for managing the data held in an Array. Arrays are a very useful tool in programming, but are much more of a rich type in haXe in terms of usability and capability, thanks to several features adopted from functional programming philosophies. As you may have suspected, haXe Arrays are subject to typing rules, meaning that an Array may only contain values that are of the same type, though of course, the `Dynamic` type can free you of this restraint. You specify the type of data held in an Array as you would specify the associated type of a template in C++. All you do is simply add the type of the contained data to the end of the Array type identifier and surround it using left and right angle brackets. For example, an Array of `Int` values is initialized like this:

```haXe
var intArray : Array<Int>;
```
Chapter 3: Learning the Basics

To initialize the data in an Array with literal values, you can supply a literal list contained in square brackets at the end of the Array declaration, which will pre-populate the Array with the given values:

```
var intArray : Array<Int> = [3,2,1];
```

When using an Array, you do not need to specify its size prior to use, as the platforms that will use the compiled Array are dynamic in length. This means that you can add new items to an Array without worrying about running out of allocated memory.

There are several types of Array in haXe: Array, List, Hash, and IntHash. Each Array type offers a slightly varied list of functionality over the other types, but all will prove their purpose in your haXe applications.

### Arrays

Arrays are the primary type for storing lists of values and provide the easiest methods for adding and extracting the values contained in the Array. Much of the syntax for haXe Arrays are similar to that of Java and ActionScript, so developers with knowledge of these languages should be able to scan this section for the minor changes.

Before using an Array in haXe, it must be instantiated, unless you initialize the Array with a literal list. This is discussed in more detail in Chapter 5, “Delving Into Object-Oriented Programming.” However, for the purposes of this chapter, you simply pass `new Array();` to the Array variable before working with it:

```
var myArray : Array<Int> = new Array();
```

You do not need to re-specify the contained data type in the instantiation, and indeed, any type identifier added to the second Array keyword is discarded.

*You can empty an Array of all data by re-instantiating it. However, in order to change the data type associated with an Array, a new Array object will need to be declared and instantiated.*

Once you have your instantiated Array object, you can start filling it with data and using it. Table 3-3 details the available methods used to manipulate Arrays.

### Table 3-3

<table>
<thead>
<tr>
<th>Array Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>length : Int</code></td>
<td>Returns the current number of items stored in the Array; length is a read-only property of the Array object.</td>
</tr>
<tr>
<td><code>concat(a : Array&lt;T&gt;) : Array&lt;T&gt;</code></td>
<td>Joins an Array to the end of the current Array object.</td>
</tr>
<tr>
<td><code>copy() : Array&lt;T&gt;</code></td>
<td>Returns a new Array structure containing the same layout and data.</td>
</tr>
</tbody>
</table>

*Table continued on following page*
### Part I: The Core Language

<table>
<thead>
<tr>
<th>Array Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert(pos : Int, x : T) : Void</code></td>
<td>Provides a means to insert data into any location of the Array.</td>
</tr>
<tr>
<td><code>iterator() : Iterator&lt;Null&lt;T&gt;&gt;</code></td>
<td>Returns an iterator of Array values.</td>
</tr>
<tr>
<td><code>join(sep : String) : String</code></td>
<td>Returns a string value containing a string representation of each of the Array items joined by a given delimiter.</td>
</tr>
<tr>
<td><code>pop() : Null&lt;T&gt;</code></td>
<td>Removes the last element of the Array and returns it.</td>
</tr>
<tr>
<td><code>push(x : T) : Int</code></td>
<td>Adds a given item to the end of the Array and returns its location.</td>
</tr>
<tr>
<td><code>remove(x : T) : Bool</code></td>
<td>Removes the first occurrence of a given item and returns a boolean success value.</td>
</tr>
<tr>
<td><code>reverse() : Void</code></td>
<td>Returns a copy of the Array with the items reversed.</td>
</tr>
<tr>
<td><code>shift() : Null&lt;T&gt;</code></td>
<td>Removes the first item of the Array and returns it.</td>
</tr>
<tr>
<td><code>slice(pos : Int, end : Int) : Array&lt;T&gt;</code></td>
<td>Copies a specified range of items in an Array and returns it as a new Array.</td>
</tr>
<tr>
<td><code>sort(f : T -&gt; T -&gt; Int) : Void</code></td>
<td>Sorts the Array according to a given comparison function where zero is returned if ( x == y ), greater than zero if ( x &gt; y ) and less than zero if ( x &lt; y ).</td>
</tr>
<tr>
<td><code>splice(pos : Int, len : Int) : Array&lt;T&gt;</code></td>
<td>Removes a given range of items from an Array and returns them as a new Array.</td>
</tr>
<tr>
<td><code>toString() : String</code></td>
<td>Returns a string representation of the Array.</td>
</tr>
<tr>
<td><code>unshift(x : T) : Void</code></td>
<td>Adds a given item to the start of the Array.</td>
</tr>
</tbody>
</table>

The **T** type in Table 3-3 is not a type in itself, but a placeholder, and represents a type value decided at run time, which enables strict typing, while remaining anonymous for any type you want to pass to the function. The **T** type is explained fully later in the book.

**Adding and Removing Items from an Array**

There are numerous ways of adding and removing items from an Array. Each method provides a way of manipulating the Array from a specific point: either at the beginning, the end, or somewhere in the middle of the Array. If you know the location of the item you want to access, you can specify that location directly and modify it as you would any variable. You do this using the index operators `[` and `]`:

```javascript
myArray[3] = 44;
var myInt = myArray[3];
```
Chapter 3: Learning the Basics

Here, the index location 3, which is the fourth item in the Array, is set to the value 44, and then that same value is passed to the variable myInt. Of course, a problem arises if the fourth item in myArray already contained a value that you didn’t want overwritten. In this case, you could use the length property of the Array, which returns the number of items in the Array:

```cpp
var numItemsInArray : Int = myArray.length;
myArray[numItemsInArray] = 44;
```

As the items in the Array start at the location 0, the length of the Array is the same value as the next available space in the Array. This means that as long as you are adding values to the location specified by the Array length, you would never overwrite existing values.

### Pushing and Popping Values

Another way to perform the same trick is to use the push method. The push method ensures that the value you wish to add to the Array is only appended to the very end of the Array:

```cpp
var lastLocation : Int = myArray.push(44);
```

The push method returns the location of the last added value, which you can then use to further manipulate the value if necessary. In the same manner, you can remove the very last item in the Array using the pop method:

```cpp
var poppedValue : Int = myArray.pop();
```

Once the last item is removed, the method returns the popped value rather than disregard it, in case you want to deal with the value in some way.

### Shifting and Unshifting Values

haXe Arrays also provide a means to add values to the start of the Array, by using the unshift method:

```cpp
myArray.unshift(44);
```

The unshift method behaves in the same manner as the push method with the exception that Void is returned, instead of the location id of the added value. This is because the location will always be 0. When adding values to the beginning of an Array, all other values contained in the Array are moved up by one, so the previous starting value will now exist at location 1.

The equivalent pop method for removing and returning values from the beginning of an Array is called shift:

```cpp
var shiftedValue : Int = myArray.shift();
```

### Adding and Removing Values from the Middle of an Array

Two main methods are provided by haXe for removing values from the middle of an Array: remove and splice. Both methods perform the item removal in very different ways, and are equally useful.
Part I: The Core Language

Remove()

The `remove` method is used to seek out and remove an item with a given value. The method takes only one parameter: the value to match and remove. Starting at the beginning of the Array, the method searches for the given value. When it finds it, the search stops and the value is removed causing all items that exist after it in the Array to move down by one location:

```haXe
var isFound : Bool = myArray.remove(4);
```

The `remove` method returns `true` if an item is found matching the given criteria and `false` if it doesn’t. Therefore, it can be possible to repeat the search, if necessary, until `false` is returned and all items with the given value are removed.

Splice()

The `splice` method allows the removal of multiple items in an Array. `splice` takes two parameters: the starting location of the chunk to remove and the number of items in the chunk:

```haXe
var chunk : Array<Int> = myArray.splice(2,2);
```

Once removed, the chunk of items is returned, which you can then store in a new Array, if you so want.

It is also possible to specify the starting location from the end of the Array using a negative number. In such a case, the value of `-1` represents the last item in the Array, while decrementing that number will count backward along the list of Array items:

```haXe
var chunk : Array<Int> = myArray.splice(-2,2);
```

Here, the last two items in the Array were removed and stored in the Array `chunk`.

Inserting Values with Insert()

Inserting a value into the middle of an Array can be accomplished using the simple `insert` method. `insert` accepts two parameters: the location to add the value and the value you want inserted. Once you have inserted the value, all items that exist after the specified location are moved up by one position:

```haXe
myArray.insert(4, someValue);
```

Copying and Concatenating Arrays

Arrays are objects and not of the haXe base types list. Because of this, passing an Array from one variable to another will not create a duplicate of the Array, but will merely create two variables that point to the same information in memory. This means that modifying one Array will automatically modify the other Array. You do not want to get into this too deeply for now, as this is the subject of a whole other chapter, but needless to say, if you want to duplicate the structure of an Array into a new Array, you will need some help. Thankfully, haXe provides you with some tools to do just that, in the guise of `slice` and `copy`.

Slice()

The `slice` method takes two parameters and returns a chunk of the Array, which just happens to look and work very similarly to the `splice` method. You have to admit that it is often confusing to discern the differences between the `slice` and `splice` methods. The two together sound like some gory horror
movie, which is what you will be left with if you use `splice` when you meant to preserve the data in your Array.

There is one difference between these two methods, however, which may sometimes raise a compiler error if you happened to use one over the other by mistake, but will at least help you to mentally differentiate the two methods. Whereas `splice` requires that you pass the length of the chunk of data to remove as the second parameter, `slice` requires that the second parameter specify the location of the end of the chunk you want to copy. This ending location is not included in the copied chunk of the Array.

Like `splice`, you can opt to use a negative value as the starting location for your chunk, but you can also use a negative value for the end of the chunk. Remember, though, that the starting location must exist before the end location:

```javascript
var arrayCpy : Array<Int> = myArray.slice(-4,-2);
```

**Copy()**

The `copy` method is applied to the entire Array, creating an exact duplicate, and takes no parameters:

```javascript
var arrayCpy : Array<Int> = myArray.copy();
```

Be warned that the `copy` and `slice` methods do not create duplicates of the contained data. Therefore, if your original Array contains objects, the data of the copied Array will contain references to those objects, not duplicates.

Now that you can copy Arrays, what about copying one Array to the end of another? This is known as concatenation, and can be applied to Arrays using the `concat` method.

**Concat()**

The `concat` method copies the structure of the given Array to the Array that calls the method. The items in the concatenated Array are not duplicated, but passed by reference, though this is not an issue if the values are of a base type such as an `Int` or `String`:

```javascript
myArray.concat(newArray);
```

**Multidimensional Arrays**

Multidimensional Arrays, while sounding like something out of Star Trek, provide a very useful feature in haXe. Otherwise know as a matrix (matrices in plural), multidimensional Arrays can help deal with groups of groups, or groups of groups of groups... you get the picture.

To better understand multidimensional Arrays, it helps to imagine them when applied to numbers. For example, a thousand contains ten hundreds, hundreds contain ten tens, and ten contains ten units. You could put this in an Array format by placing each of the four numerical groups into Arrays of their own, like this:

```javascript
var thousandths : Array<Int> = [0,1,2,3,4,5,6,7,8,9];
var hundredths : Array<Int> = [0,1,2,3,4,5,6,7,8,9];
var tens : Array<Int> = [0,1,2,3,4,5,6,7,8,9];
var units : Array<Int> = [0,1,2,3,4,5,6,7,8,9];
```
The problem with this method is that the groups held in each of the arrays are not joined. You only have one of each Array type, while in theory, you should have ten thousandths Arrays, one hundred hundredths Arrays, one thousand tens Arrays, and ten thousand units Arrays. What's more, each of the Arrays should be linked to the relevant items in the parent Arrays.

To resolve this issue, you can create Arrays of Arrays by specifying the type of a parent Array as an Array, and repeat this for each child Array in the hierarchy:

```javascript
var numbers : Array<Array<Array<Array<Int>>>>;  
```

Once this is done, you need to instantiate each of the Arrays in the hierarchy:

```javascript
numbers = new Array();
for ( a = 0...9 )
{
    numbers[a] = new Array();
    for ( b = 0...9 )
    {
        numbers[a][b] = new Array();
        for ( c = 0...9 )
        {
            numbers[a][b][c] = new Array();
            for ( d = 0...9 )
            
                numbers[a][b][c][d] = ( a * 1000 ) + ( b * 100 ) + ( c * 10 ) + d;
        }
    }
}
```

This should continue until you have instantiated all the Arrays in the hierarchy, though you can opt to do this as you require them, rather than doing it all upfront. Once this is done, you can then access the Arrays by supplying one pair of index operators per Array:

```javascript
var thousands : Int = 6;
var hundreds : Int = 4;
var tens : Int = 7;
var units : Int = 3;
numbers[thousands][hundreds][tens][units] = 6473;
```

or:

```javascript
numbers[6][4][7][3] = 6473;
```

Theoretically, a multidimensional Array can have as many dimensions as you like, though it is very rare to use more than two dimensions at the time.

**Lists**

Lists are the most lightweight of all the Array types. While they are similar to the Array type, they use less CPU processing power and are smaller in terms of memory requirements when used with Neko. This is because the List type provides less of the heavier functionality such as directly accessing the values it contains using indexes. Instead, the List must be examined in a loop using an iterator or...
through one of its available methods, as trying to access the indexes of a List will result in a compiler error. Loops and iterators are covered in the next chapter.

Surprisingly, the List type supports quite a differing assortment of methods over the Array type. The List’s primary use is as a container for objects that are not wholly independent of one another, and whose values are only of use when read or manipulated as a group. To this end, the methods of the List type are oriented toward batch processing and iteration of its contained values. While the length and remove methods exist in the List type in the same fashion as the Array, some methods are altogether different.

Table 3-4 lists the methods available to a List object.

Table 3-4

<table>
<thead>
<tr>
<th>List Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length : Int</td>
<td>Returns the number of items in the List; it is read-only.</td>
</tr>
<tr>
<td>add( item : T ) : Void</td>
<td>Adds an item to the end of the List.</td>
</tr>
<tr>
<td>clear() : Void</td>
<td>Empties the List of all items.</td>
</tr>
<tr>
<td>filter( f : T -&gt; Bool ) : List&lt;T&gt;</td>
<td>Returns a List filtered with a given function f where f(x) = true.</td>
</tr>
<tr>
<td>first() : T</td>
<td>Returns the first item in a List.</td>
</tr>
<tr>
<td>isEmpty() : Bool</td>
<td>Returns true if the List is empty, otherwise false is returned.</td>
</tr>
<tr>
<td>iterator() : Iterator&lt;T&gt;</td>
<td>Returns an iterator of the items in the List.</td>
</tr>
<tr>
<td>join( sep : String ) : String</td>
<td>Returns a string representation of all the items in the List joined by a given delimiter string.</td>
</tr>
<tr>
<td>last() : T</td>
<td>Returns the last item in a List.</td>
</tr>
<tr>
<td>map&lt;X&gt; ( f : T -&gt; X ) : List&lt;X&gt;</td>
<td>Returns a new List where all items have been modified by a given function.</td>
</tr>
<tr>
<td>pop() : T</td>
<td>Removes the first item in a List and returns it.</td>
</tr>
<tr>
<td>push( item : T )</td>
<td>Adds an item to the beginning of the List.</td>
</tr>
<tr>
<td>remove( v : T ) : Bool</td>
<td>Removes the first item equal to a given value. Returns true if an item is found; otherwise false is returned.</td>
</tr>
<tr>
<td>toString() : String</td>
<td>Returns a string representation of the List.</td>
</tr>
</tbody>
</table>
Part I: The Core Language

Adding and Removing Items from a List

Adding values to a List is very much a different affair to an Array. As indexing is not supported, one cannot simply pass a value to a new index. To alleviate this difference, the List type provides an add method that facilitates adding a value to the next available index in a similar fashion to the Arrays push method. The differences between the two are that the add method returns Void instead of the index id where the data was added:

```haxe
var myList : List<Int> = new List();
myList.add(someValue);
```

You can also add a value to the beginning of a List, using the push method. However, this is where things start to get pretty confusing, as it bears the same name as the push method used for adding a value to the end of an Array.

```haxe
myList.push(someValue);
```

As with Arrays, the push method also has a paired pop method, which removes and returns the item at the beginning of the List:

```haxe
var poppedValue : Int = myList.pop();
```

Unlike the Array, the List has no method to directly remove the item stored at the end of the List.

Querying Values in a List

As noted earlier, Lists do not support indexes, so you cannot directly access an item within the List. However, the List does provide a couple of methods that will enable you to read the values stored at the very beginning or end of the List without removing the items. These are first and last.

As you would expect, the method first allows you to query the first item in a List, while last allows you to query the last item:

```haxe
var firstItem : Int = myList.first();
var lastItem : Int = myList.last();
```

Hash Tables

A Hash, or Hash Table, is a type of Array that allows access to its values using keys. Storing data in an Array of this type removes the ambiguity of the contained data and helps to give some form of individuality to each item. The type of key used by the Hash depends on the type of Hash you choose to use. haXe offers two versions of Hash that provide identical functionality, except the Hash type uses keys of a String type, while the IntHash uses keys of an Int type.

Like Lists, Hash's do not support accessing values using indexes. This is because Hash's do not store their values in such an obviously linear fashion. One might consider a Hash to resemble more closely that of an object in JavaScript and Actionscript, with the added benefit that one can iterate through each of the values stored in a Hash.
Table 3-5 defines the methods provided by the Hash and IntHash classes.

**Table 3-5**

<table>
<thead>
<tr>
<th>Hash Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exists( key : String ) : Bool</td>
<td>Returns true if a value exists for a given key</td>
</tr>
<tr>
<td>get( key : String ) : T</td>
<td>Returns the value for a given key</td>
</tr>
<tr>
<td>iterator() : Iterator&lt;T&gt;</td>
<td>Returns an iterator of the item values in the</td>
</tr>
<tr>
<td></td>
<td>Hash or IntHash</td>
</tr>
<tr>
<td>keys() : Iterator&lt;String&gt;</td>
<td>Returns an iterator of the item keys in the</td>
</tr>
<tr>
<td></td>
<td>Hash or IntHash</td>
</tr>
<tr>
<td>remove( key : String ) : T</td>
<td>Removes an item specified by the given key</td>
</tr>
<tr>
<td>set( key : String, value : T) : Void</td>
<td>Sets the value of a given key</td>
</tr>
<tr>
<td>toString() : String</td>
<td>Returns a string representation of the Hash or</td>
</tr>
<tr>
<td></td>
<td>IntHash</td>
</tr>
</tbody>
</table>

**Adding, Querying, and Removing Items from a Hash**

The methods in a Hash type are not extensive. Unlike the Array, you are not offered countless methods for every possible undertaking you could possibly imagine. Instead, you are offered simple methods for basic procedures. For example, to add an item to a Hash, you use the set method:

```javascript
var myHash : Hash<String> = new Hash();
myHash.set("newKey", "someValue");
```

If you then want to retrieve the value of this key, you would use the get method:

```javascript
var myValue = myHash.get("newKey");
```

You can then remove the key and its paired value from the Hash using remove:

```javascript
myHash.remove("newKey");
```

Once you remove your key, querying the key again results in a null value. In order to check that a key exists before you query it, you can use the exists method:

```javascript
var doesExist : Bool = myHash.exists("newKey");
```

If the key exists, the method will return true. Otherwise, false is returned.
Part I: The Core Language

To get a better feel for the Hash type, enter the following code and save it as HashTest.hx:

```haxe
class HashTest
{
    static public function main()
    {
        var myHash : Hash<Int> = new Hash();
        myHash.set("one", 1);
        myHash.set("two", 2);
        myHash.set("three", 3);
        trace(myHash.get("three"));
            // Outputs: 3
        trace(myHash.get("two"));
            // Outputs: 2
        trace(myHash.get("one"));
            // Outputs: 1
    }
}
```

Having created the Hash, you then used the set method to add three items to the Hash object using string identifiers, or keys. Next, you queried the Hash object using the same keys using the get method, which retrieved the associated values.

Using Dates in haXe

The Date type is not considered as part of the standard types in haXe. However, haXe does support dates and actually has quite an extensive list of tools for working with them. So extensive, in fact, that the methods used for dealing with dates are spread across two classes to avoid bloating applications that use them.

Table 3-6 provides a list of the methods available in the Date class.

**Table 3-6**

<table>
<thead>
<tr>
<th>Date Method</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDate() : Int</td>
<td>Returns the day of the month as 1–31</td>
</tr>
<tr>
<td>getDay() : Int</td>
<td>Returns the day of the week; 0 = Sun, 1 = Mon . . .</td>
</tr>
<tr>
<td>getFullYear() : Int</td>
<td>Returns the full year, i.e., 1976</td>
</tr>
<tr>
<td>getHours() : Int</td>
<td>Returns the hour in a 24-hour format; 0–23</td>
</tr>
<tr>
<td>getMinutes() : Int</td>
<td>Returns the number of minutes; 0–59</td>
</tr>
<tr>
<td>getMonth() : Int</td>
<td>Returns the month; 0 = Jan, 1 = Feb . . .</td>
</tr>
<tr>
<td>getSeconds() : Int</td>
<td>Returns the number of seconds; 0–59</td>
</tr>
</tbody>
</table>
### Date Method

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getTime()</code> : Float</td>
<td>Returns the number of milliseconds since 1st Jan 1970</td>
</tr>
<tr>
<td><code>toString()</code> : String</td>
<td>Returns a string representation of the Date object</td>
</tr>
<tr>
<td><code>static fromString</code></td>
<td>Instantiates a Date object from a string representation of a date.</td>
</tr>
<tr>
<td><code>static fromTime</code></td>
<td>Instantiates a Date object from an integer depicting the number of milliseconds since 1st Jan 1970</td>
</tr>
<tr>
<td><code>static now()</code> : Date</td>
<td>Instantiates a Date object using the current date and time</td>
</tr>
</tbody>
</table>

---

### Creating a Date Object

The primary class used for working with dates is the `Date` class. Like Arrays, this class needs to be instantiated before you can use it. However, this instantiation is a little longer than Array instantiation as it expects a whole number of parameters. The definition for instantiating a `Date` object looks like this:

```javascript
var year : Int = 2007;
var month : Int = 9;
var day : Int = 13;
var hours : Int = 23;
var mins : Int = 39;
var secs : Int = 30;
var date = new Date( year, month, day, hours, mins, secs );
```

Long, huh? The instantiation requires all the parameters to be set, including the time, though if it’s only the date you’re after, you can supply a zero for each of the time related parameters. Here is an example showing how a `Date` object might be instantiated:

```javascript
var birthDate : Date = new Date( 1976, 10, 3, 13, 15, 0 );
```

This example instantiates the variable `birthDate` with a date and time: the 3rd of November 1976 at 13:15 in the afternoon. As you can see, the value supplied for month has been set to 10, rather than 11 for November. This is not an error. Unfortunately, because of ECMA specification, the month value starts at 0, rather than 1, so this value alone must be set one less than the expected value.

Luckily, you won’t have to provide this kind of instantiation every time you require a date object. For example, if it’s the current date and time you’re after, the `Date` class provides a `now` method that initializes the date for you with the current date and time:

```javascript
var birthDate : Date = Date.now();
```
You can also create a Date object from a string or float using `fromString` or `fromTime`, respectively. The string instantiation requires that a certain format be met containing either the date, time or date and time using the format YYYY-MM-DD to represent the date and HH:MM:SS for the time. So, to initialize a date with a birthday using the string method, you would do the following:

```javascript
var birthDate : Date = Date.fromString("1976-11-03 13:15:00");
```

This time, the month value must start from 1, so would be the normal value for that month.

Using the `fromTime` method is a little less useful in that it requires you to pass the number of milliseconds that have elapsed from the 1st of January 1970 to the date you wish to represent. This is doubtful something you will be able to pull out of the air, but is a great tool when re-instantiating a numerically serialized version of a date and time:

```javascript
var birthDate : Date = Date.fromTime(215874900000);
```

### Retrieving the Components of a Date

Once you have your date all nicely snug and warm in a Date object, the chances are you’re going to want to query certain components of that date. By component, it is meant a unit of time, such as the day of the month or the month itself. haXe provides a list of functions for retrieving these details in a numerical format. What you do with that data when you have it is up to you.

The Date class supplies two methods that provide you with a way of extracting the day from a Date object: `getDay` and `getDate`. You’ll find that these two methods aren’t terribly descriptive, as they both return the day component of your Date object. However, while the method `getDay` returns the day of the month as an integer between 1 and 31, `getDate` returns the day of the week as an integer between 0 and 6, where 0 is Sunday, 1 is Monday, 2 is Tuesday and so on:

```javascript
var dayOfMonth : Int = birthDate.getDate();  // 1 ... 31
var dayOfWeek : Int = birthDate.getDay();    // 0 = Sunday, 1 = Monday ...
```

If you apply both methods to the birthday, you will receive 3 from the call to `getDate` and 3 from the call to `getDay`, as the date was a Wednesday.

Extracting the month and year components of a Date object is a little more foolproof. To help you with these tasks, you are provided with the aptly named `getMonth` and `getFullYear` methods. `getMonth` returns the month of the year as an integer from 0 to 11, with 0 representing January and so on, much like the value required when instantiating the Date object. The `getFullYear` method returns the year as it is written, so for this birthday, it would return 1976. It is named thus to emphasize that the method returns a full four-digit representation of the year:

```javascript
var month : Int = birthDate.getMonth();    // 0 = January, 1 = February ...
var year : Int = birthDate.getFullYear();  // 1976 etc.
```
Chapter 3: Learning the Basics

haXe provides you with the functions `getHours`, `getMinutes`, and `getSeconds` for retrieving the time components of a Date object as integer values. The returned values represent the time components, as they would be written on a 24-hour digital clock:

```haxe
var hours : Int = birthDate.getHours(); // 0 ... 23
var mins : Int = birthDate.getMinutes(); // 0 ... 59
var secs : Int = birthDate.getSeconds(); // 0 ... 59
```

The last method provided by the Date object is the `getTime` method. `getTime` returns the number of milliseconds elapsed since the 1st of January, 1970, which is the exact data required by the Date object's `fromTime` method:

```haxe
var time : Float = birthDate.getTime(); // 215874900000 for the birthday
```

**The DateTools Class**

Unfortunately, haXe offers only two functions for manipulating the date and time of a Date object: `delta` and `format`. Also, the function `getMonthDays` is available for simple day count calculations.

**delta()**

It is expected that any adjustments to a Date object require the use of a secondary Date object or custom functions to aid in performing date and time calculations. This function, `delta`, forms part of a new class called `DateTools` and accepts the Date object you want to modify and the number of milliseconds, as a Float value, that you wish to add to the date. For example, if you realized from the birth notes that the actual time was actually 13:20 in the afternoon, you could add five minutes to the date by doing the following:

```haxe
birthDate = DateTools.delta(birthDate, 300000.0);
```

If, however, the time was actually 13:10 in the afternoon, then you could remove those five minutes by supplying a negative value.

When dealing with larger dates, it is advisable to create your own functions for working out the number of seconds in a month, day, hour, and so on. You look at how haXe deals with functions in the next chapter.

**format()**

The `format` function provides the reverse functionality of the Date.fromString() method, whereby you supply the Date object and a string representing the format of the date you want to output, and the function will return a string containing your newly processed date. The `format` string will accept any values as specified in the strftime standard, with the exception of month names because of internationalization issues. Table 3-7 details the accepted string tokens used to determine the output format.
### Part I: The Core Language

#### Table 3-7

<table>
<thead>
<tr>
<th>Token</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%a</code></td>
<td>Abbreviated weekday name (e.g., Fri)</td>
</tr>
<tr>
<td><code>%A</code></td>
<td>Full weekday name (e.g., Friday)</td>
</tr>
<tr>
<td><code>%b</code></td>
<td>Abbreviated month name (e.g., Oct)</td>
</tr>
<tr>
<td><code>%B</code></td>
<td>Full month name (e.g., October)</td>
</tr>
<tr>
<td><code>%c</code></td>
<td>The standard date and time string</td>
</tr>
<tr>
<td><code>%d</code></td>
<td>Day of the month, as a number (1–31)</td>
</tr>
<tr>
<td><code>%H</code></td>
<td>Hour, 24-hour format (0–23)</td>
</tr>
<tr>
<td><code>%I</code></td>
<td>Hour, 12-hour format (1–12)</td>
</tr>
<tr>
<td><code>%j</code></td>
<td>Day of the year, as a number (1–366)</td>
</tr>
<tr>
<td><code>%m</code></td>
<td>Month as a number (1–12)</td>
</tr>
<tr>
<td><code>%M</code></td>
<td>Minute as a number (0–59)</td>
</tr>
<tr>
<td><code>%p</code></td>
<td>Locale’s equivalent of A.M. or P.M.</td>
</tr>
<tr>
<td><code>%S</code></td>
<td>Second as a number (0–59)</td>
</tr>
<tr>
<td><code>%U</code></td>
<td>Week of the year (0–53), where week 1 has the first Sunday</td>
</tr>
<tr>
<td><code>%w</code></td>
<td>Weekday as a decimal (0–6)</td>
</tr>
<tr>
<td><code>%W</code></td>
<td>Week of the year (0–53), where week 1 has the first Monday</td>
</tr>
<tr>
<td><code>%x</code></td>
<td>Standard date string</td>
</tr>
<tr>
<td><code>%X</code></td>
<td>Standard time string</td>
</tr>
<tr>
<td><code>%y</code></td>
<td>Year in decimal, without the century (0–99)</td>
</tr>
<tr>
<td><code>%Y</code></td>
<td>Year in decimal, with the century</td>
</tr>
<tr>
<td><code>%z</code></td>
<td>Time zone name %% a percent sign</td>
</tr>
</tbody>
</table>

**getMonthDays()**

`getMonthDays` accepts an instantiated `Date` object and returns the number of days available for the set month. Therefore, if you were to pass the `birthDate` object to `getMonthDays`, it would return the value 30.
Manipulating Data

Okay, so now you know how to represent basic data types in your applications, but what can you do with that data? Well, some applications that deal with basic data types supply some form of processing algorithms that modify the data in some way. This might be some simple math for calculating values or perhaps assigning a template for laying out character data. The haXe standard library contains a large number of methods that can provide manipulative functionality on basic types, and you will be covering many of these over the next few pages.

Operators

Before you delve into the haXe standard library, it is important that you understand the use of operators. You will not cover the use of operators with too much depth as prior knowledge of their use is a prerequisite for this book. However, many of you reading this book may very well come from an HTML or Flash development background, and so you may not have a complete understanding of all the available operators at your disposal. Therefore, there is an attempt to summarize their usage, though those of you already very familiar with operator usage in languages such as C++, Java, JavaScript, or ActionScript, please feel free to skip ahead.

Table 3-8 provides an overview of the operators usable in the haXe language.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associative Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Left to right</td>
<td>Grouping operators</td>
</tr>
<tr>
<td>!</td>
<td>Right to left</td>
<td>Negation</td>
</tr>
<tr>
<td>++</td>
<td>Left to right</td>
<td>Increment</td>
</tr>
<tr>
<td>--</td>
<td>Left to right</td>
<td>Decrement</td>
</tr>
<tr>
<td>*</td>
<td>Left to right</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Left to right</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Left to right</td>
<td>Modulo</td>
</tr>
<tr>
<td>+</td>
<td>Left to right</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Left to right</td>
<td>Subtraction</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Right to left</td>
<td>Left shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Right to left</td>
<td>Right shift</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left to right</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Left to right</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Left to right</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Left to right</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>

Table continued on following page
Part I: The Core Language

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associative Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Left to right</td>
<td>Equality</td>
</tr>
<tr>
<td>!=</td>
<td>Left to right</td>
<td>Inequality</td>
</tr>
<tr>
<td>&amp;</td>
<td>Left to right</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Left to right</td>
</tr>
<tr>
<td>^</td>
<td>Left to right</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Left to right</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>= *= /= %= += -= &amp;= ^=</td>
<td>=</td>
<td>Right to left</td>
</tr>
</tbody>
</table>

These operators have been put into the order of precedence with the higher precedence operators starting at the top of the list.

**Binary Operators**

Binary operators are so named because they operate on two values. They include the addition operator (+) for adding values together, the subtraction operator (-) for subtracting one value from another, the forward slash operator (/) for dividing one value by another, the asterisk operator (*) for multiplying two values, the modulo operator (%) for finding the remaining value of a division and the equals operator (=), which is otherwise known as the assignment operator, and is used to assign values to variables. These are the core operators that you should know well, regardless of your programming background.

**Operator Precedence**

When performing regular arithmetic with literal or variable data, it is important to bear in mind the order of operator precedence. For example, if you were to try to execute the following arithmetic, what result would you get?

```swift
var result:Float = 10 * 2 + 1;
```

Without understanding the order or operator precedence, you may multiply the 10 by 2, which would be 20, then add the one, making 21. You could also opt for a different route and add the 2 and the 1 to make 3, then multiply by 10 to get 30. With operator precedence, however, the order states that the * operator will be executed before the + operator. Therefore, the result should be 21. Likewise, if you examine the next equation, what would you expect the result to be?

```swift
var result:Float = 21 * 2 + 5 - 4 / 2;
```

The order of precedence dictates that the first part of the equation that will be calculated is 21 * 2. Next, the equation 4 / 2 is calculated. The answer to the first equation is then added to 5, which will equate to 47 and the second equation is then subtracted from this value, making 45. As you can see, even when you know the order of precedence of operators, it can still be difficult to immediately see how the equation should be tackled.
To aid you in producing formulas that are easier to discern, you could use the grouping operators. As the grouping operators are always the first operators to be evaluated, this can enable you to force an order of calculation while providing a more pleasing way to present your data. Using the grouping operators, you can rewrite the previous problem like this:

```javascript
var result:Float = ((21 * 2) + 5) - (4 / 2);
```

You are now left with a formula that is more pleasing to the eye and should be readable by those not familiar with operator precedence. At the same time, you could decide to change the order of precedence by moving the grouping operators and not the binary operators or values, such as:

```javascript
var result:Float = (21 * ((2 + 5) - 4)) / 2;
```

The equation now reads as $2 + 5 = 7 \ - \ 4 = 3 \ * \ 21 = 63 \ / \ 2 = 31.5$. An entirely different result, altogether.

The Modulo Operator

One of the binary operators you may not be familiar with is the modulo operator ($\%$). The purpose of the modulo operator is to determine the remainder of a division. For example, if you wanted to find the remainder of 10 divided by 3, you could use the expression:

```javascript
remainder = 10 % 3;
```

While it may not seem very useful at first, the modulo operator can come in very handy when trying to determine if a value is exactly divisible by another value. For example, suppose you wanted to repeat a task, but carry out a specific function for every fifth repetition. The best trick would be to create a variable and initialize it with zero, and then every time the task repeated, you could increase the variable by one and check its value using modulo, like so:

```javascript
repeater = repeater + 1;
if (repeater % 5 == 0) myFunction();
```

If the value returned by this equation is zero, then you would know to execute your function. Useful, huh?

Increment and Decrement

Very often in your code, you will use variables as a form of counting. When counting, you would often start with a number and either increase or decrease its value by one. So, if you have a variable called `count` and wish to count up, you could do the following:

```javascript
++count;
```

Now, this is all well and good, but it does take up quite a bit of real estate. You can, however, simplify the way this task is executed by using the increment (++) operator, which increases a value by one, or decrement (--) operator, which decreases a value by one. To use this feature, you must append the operator either before or after the value:

```javascript
count++;
++count;
```
You can also use the increment and decrement operators within an expression. Here, the position of the operator affects when the value is changed. By applying the operator after the variable, the value is not altered until after the expression is executed, while applying the operator before the variable alters the value before the expression is executed. Let's try this in an example. Type the following code and save it in a file called Increment.hx:

```haxe
class Increment
{
    public static function main()
    {
        var myVal:Int = 0;
        trace(myVal);
        // Outputs: 0
        trace(++myVal);
        // Outputs: 1
        trace(myVal--);
        // Outputs: 1
        trace(myVal);
        // Outputs: 0
    }
}
```

Once the integer was initialized, its value of 0 was simply displayed to the screen. Then, the variable was displayed on the screen a second time while incrementing the value. As the increment operator is before the value, the value was incremented before it was printed, so 1 was displayed. The value was then printed to screen a third time while decrementing the value. However, as the decrement value was appended after the variable, its value was not calculated until after it was printed, so 1 was displayed a second time. Finally, the current value was printed to screen, which should then be 0.

**Logical Comparison Operators**

Logical comparison operators are used to discern the differences or equality of values and expressions. Like binary operators, the logical comparison operators work with two values or expressions; one to either side of the operator. All logical comparison operators return a boolean value. `true` is returned if the comparison expression is true and `false` if the expression is false.

The most common form of logical comparison is to test whether two expressions are equal (`==`) or not equal (`!=`):

```haxe
var myBoolA : Bool = (1 != 2);
var myBoolB : Bool = (1 == 2);
```

Here, `myBoolA` will be `true`, as 1 is not equal to 2, which will mean the expression for `myBoolB` (1 is equal to 2) will be `false`.

You can also test to see if expressions are greater than (`>`), greater than or equal (`>=`), less than (`<`), or less than or equal (`<=`). For example:

```haxe
var myBoolA : Bool = (1 < 2);
var myBoolB : Bool = (1 >= 2);
```
In this example, `myBoolA` will be `true`, as 1 is smaller than 2, but `myBoolB` will be `false`, as 1 is not greater or equal to 2.

**Logical AND and OR**

If you need to compare the values of more than one logical comparison, you can join the comparisons using the logical AND (`&&`) or logical OR (`||`) operators. The logical AND will only return `true` if both expressions on either side are true, while the logical OR will return `true` if either of the expressions are true:

```haXe
var myBoolA : Bool = (1 < 2) && (2 == 2);
var myBoolB : Bool = (2 >= 4) || (3 < 4);
```

In this example, `myBoolA` will be `true`, as 1 is less than 2 and 2 is equal to 2. The logical AND operator required both expressions to be true, so it was able to return `true`. `myBoolB` is also `true`, as although 2 is not greater than or equal to 4, 3 is less than 4, so the OR operator will return `true`.

An issue arises if you wish to stack more than two expressions using the logical AND and OR operators. Looking at the next example, see if you can guess what the outcome should be.

```haXe
var myBool : Bool = (1 < 2) && (2 == 2) && (2 >= 4) || (4 < 3);
```

If you guessed correctly, you should have expected `false` to be the answer. The reason for this is that, when grouping logical operators, any adjacent expressions joined by AND operators will all have to return `true` for the overall result to be `true`, while adjacent OR operators will only require one expression to return `true`. To see this more clearly, it helps to picture the expressions enclosed within the group operators (`(` and `)`).

```haXe
var myBool : Bool = (1 < 2) && (2 == 2) && (2 >= 4) || (4 < 3);
```

As you can see, each expression joined by the logical AND operator is reliant on its neighbor, while the logical OR operator separates the left-hand expression from its right-hand neighbor.

You will look more closely at logical comparison operators when you examine the `if...else` commands in Chapter 4, “Controlling The Flow Of Information.”

**Assignment Operators**

So far, you have only seen one of the assignment operators, the `=` operator. This is known as the assignment operator. However, this is not the only assignment operator available in the haXe language.

When working with numerically typed variables, it is common to have to perform mathematical operations using other numerically typed variables or literal values and re-assign the value back into one of the numerical variables. Just as you can increase or decrease a variables value by one using the increment and decrement operators, you can also perform mathematical equations on a variable using a similarly styled set of assignment operators. For example, to increase a variables value by one, rather than using the increment operator, you could use the addition assignment operator (`+=`), like this:

```haXe
count += 1;
```
Part I: The Core Language

As you can see, this expression is very similar to assigning the value 1 to a variable, however, the addition symbol before the assignment symbol states that you want to add the 1 to the existing value of the count variable and then assign the result of that equation into count. If you want, you could use a number other than 1, such as 23 or -127, which would equate to count = count + 23 or count = count + -127, respectively.

You could also perform other mathematical equations, such as division or multiplication, by substituting the addition symbol for the relevant required operator.

**Bitwise Operators**

You do not want to get too involved in the discussion of bitwise operators as bitwise mathematics venture beyond the scope of this book, however, it is important to know these operators and the possibilities they can bring to your applications. When working through the examples in this part of the chapter remember the Neko 31-bit limit for integers and perhaps try to compile the examples in Flash where necessary.

Bitwise mathematics involves moving, altering, and comparing the bits of a given value. For those who do not know, bits are the basic building blocks of characters, numbers, and mathematical equations within a computer's memory. When discussing a simple equation such as:

\[
15 + 20 = 35
\]

You are really saying:

\[
\begin{align*}
00000000 &\quad 00000000 &\quad 00000000 &\quad 00001111 \\
+ &\quad 00000000 &\quad 00000000 &\quad 00010100 \\
= &\quad 00000000 &\quad 00000000 &\quad 00100011
\end{align*}
\]

Each of the previous lines of 1's and 0's form a pattern that represents the construction of a 32-bit integer. Moving right to left, you can assign values to each bit that is a multiplication of 2 times the previous bit value. For example, you could represent the bits of a 32-bit integer as:

\[
\begin{align*}
&2^{31} &\quad 2^{30} &\quad 2^{29} &\quad 2^{28} &\quad 2^{27} &\quad 2^{26} &\quad 2^{25} &\quad 2^{24} \\
&2^{23} &\quad 2^{22} &\quad 2^{21} &\quad 2^{20} &\quad 2^{19} &\quad 2^{18} &\quad 2^{17} &\quad 2^{16} \\
&2^{15} &\quad 2^{14} &\quad 2^{13} &\quad 2^{12} &\quad 2^{11} &\quad 2^{10} &\quad 2^9 &\quad 2^8 \\
&2^{7} &\quad 2^{6} &\quad 2^{5} &\quad 2^{4} &\quad 2^{3} &\quad 2^{2} &\quad 2^{1} &\quad 1
\end{align*}
\]

To save space, each group of 8 bits has been added to a new line. The information can then be further broken down as:

\[
\begin{align*}
2147483648 &\quad 1073741824 &\quad 536870912 &\quad 268435456 &\quad 134217728 &\quad 67108864 &\quad 33554432 &\quad 16777216 \\
8388608 &\quad 4194304 &\quad 2097152 &\quad 1048576 &\quad 524288 &\quad 262144 &\quad 131072 &\quad 65536 \\
32768 &\quad 16384 &\quad 8192 &\quad 4096 &\quad 2048 &\quad 1024 &\quad 512 &\quad 256 \\
128 &\quad 64 &\quad 32 &\quad 16 &\quad 8 &\quad 4 &\quad 2 &\quad 1
\end{align*}
\]
Chapter 3: Learning the Basics

So, to re-examine the original equation, you are actually saying:

\[
(1 + 2 + 4 + 8) \\
+ \\
(4 + 16) \\
= \\
(1 + 2 + 32)
\]

Normally, you will not have to perform math on the bits themselves as performing math with their numerical equivalents is far easier to understand and just as fast. However, by being able to manipulate the bits of a value, you can perform certain tricks that help to make your code easier to read while saving on resources.

**Shifting Bits**

Let’s look at the bits again. If you wanted, you could decide to re-implement the 32 bits of an integer and put them to another use. For example, it may be that you have four integers you wish to represent in your application that will never exceed the value 255. Normally, you would simply create four integer variables and maintain them, but if the four values are grouped by functionality, it seems silly to waste all that extra memory and code by doing so. Instead, you could mentally partition a single integer value into four segments, so instead of the previous bit representation, you could envisage the bits as:

```
  128 64 32 16 8 4 2 1
  128 64 32 16 8 4 2 1
  128 64 32 16 8 4 2 1
  128 64 32 16 8 4 2 1
```

Here, you are still using the same number of bits, only preparing yourself to treat it as four separate values. The problem remaining, then, is how to store your values into the new structure. Take a look at your integers:

```
Int 1 = 155: 00000000 00000000 00000000 10011011
Int 2 = 80: 00000000 00000000 00000000 01010000
Int 3 = 17: 00000000 00000000 00000000 00010001
Int 4 = 131: 00000000 00000000 00000000 10000011
```

If you simply assign your four different values to the 32-bit integer, you would be merely repeatedly overwriting each assigned value with the next. This is because your values occupy the last 8 bits of each integer, so when they are assigned to your new variable, they are simply transferred to the last 8 bits of the new integer. What you need to do instead is to shift the bits, so instead of assigning to the last 8 bits, they are assigned to the next free group of 8 bits.

The answer is to use the bitwise shift operators, which allow you to left shift (<<) or right shift (>>>) bits by a specified number of places. Therefore, to shift the group of bits four places to the left, you could write:

```java
myContainer = myBits << 4;
```
Part I: The Core Language

This is the equivalent of:

\[
\begin{align*}
00000000 & \ 00000000 & \ 00000000 & \ 01010000 << 4 \\
= & \ 00000000 & \ 00000000 & \ 00000101 & \ 00000000
\end{align*}
\]

As you can see, when shifting your bits left, zeros were added to the right of your value to keep the number of bits to 32. Had there been 1’s occupying the last 4 bits on the left, they would have been lost in the shift. Shifting your bits left alters your value by making it larger. In fact, for each place the value is shifted left, it is multiplied by 2. Shifting right then has the effect of dividing your value by two.

You can combine your shifted integers by simply adding them together, so each value occupies the empty space of the other values. Once you have shifted your bits and assigned them to the new variable, you can then retrieve your values by simply shifting the new variable’s value to the right and extracting them.

Filtering Bits

When trying to extract your values, you may notice a small problem. You know that when you shift your bits and add them together, you are in effect filling empty spaces, but when you then want to extract your values, you find that the individual values you started with are now corrupting each other. For example, when combining your four integers, your new variable will have looked like this:

\[
10011011 \ 01010000 \ 0010001 \ 00000111
\]

Here, each group of 8 bits represents one of your original values. If you start shifting right to extract your values, the left area of each value, with the exception of the far left integer, will be corrupted with each of the values occupying the higher bits. To combat this problem, you need to use the bitwise AND (\&) operator, which will help you filter your values from the containing variable.

haXe supports three operators for bitwise filtering. They are AND (\&), OR (\|), and XOR (\^). The AND and OR filters work similarly to the comparison equivalents, except that, instead of comparing true or false expressions and returning a resulting boolean value, they instead compare bits and return a resulting bit value. XOR, which stands for eXclusive OR, compares two bits and returns 1 only if one of the bits is a 1. If both bits are or neither bit is a 1, then XOR will return 0. This is quite different to OR, which will return a 1 if either bit or both bits are 1. AND will return a 1 if neither bit is 0. Let’s see some examples; scan the bits in each pair and try to guess what result each filter will return:

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>01001011 11011010</td>
<td>01001010 11011011</td>
<td>10010001</td>
</tr>
<tr>
<td>10111001 00011011</td>
<td>00100101 11101011</td>
<td>11010101</td>
</tr>
<tr>
<td>01100010</td>
<td>01000010 11101011</td>
<td>11010101</td>
</tr>
</tbody>
</table>

Now that you know what the filters do, you can look at ways in which to retrieve your values. If you have a 32-bit integer and wish only to retrieve the first 8 bits, you can apply the AND filter using a value that has all of the first 8 bits filled as 1 and all remaining bits filled with 0. This will force all bits after the first 8 to be discarded:
Chapter 3: Learning the Basics

10011011 01010000 00010001 10000011 & 00000000 00000000 00000000 11111111 = 00000000 00000000 00000000 10000011

You then simply shift the original value to the right by 8 bits and repeat for the next value. Applying this to your original problem, your class might look like this:

```haxe
class ShiftBits {
    public static function main() {
        var newInt:Int;
        newInt = 155 + (80 << 8) + (17 << 16) + (131 << 24);
        trace(newInt & 255);
        trace(newInt >> 8 & 255);
        trace(newInt >> 16 & 255);
        trace(newInt >> 24 & 255);
    }
}
```

Try compiling this program and see for yourself.

The Math Class

Don’t panic; it is not a discussion about school here. However, you have been looking at number values, so sooner or later, you must start doing some math with your values.

If you think back to those days at school when you were up to your neck in trigonometry and calculus, you will probably remember having to use a scientific calculator in order to work out the tough math questions involving scary math functions such as sine and cosine. In haXe, all such math functions are stored in the `Math` class.

Fortunately, this book is not a math textbook, so much of the math theory will be left out of this chapter, but it will show you some of the more common methods of the `Math` class, so you at least have some grounding with which to work. Table 3-9 details the methods of the `Math` class.

<table>
<thead>
<tr>
<th>Math Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEGATIVE_INFINITY : Float</td>
<td>Property representing a negative <code>divide by zero</code></td>
</tr>
<tr>
<td>NaN : Float</td>
<td>Property representing <code>Not a Number</code></td>
</tr>
<tr>
<td>PI : Float</td>
<td>Property representing the value of Pi</td>
</tr>
<tr>
<td>POSITIVE_INFINITY : Float</td>
<td>Property representing a positive <code>divide by zero</code></td>
</tr>
</tbody>
</table>

Table continued on following page
Part I: The Core Language

<table>
<thead>
<tr>
<th>Math Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(value : Float) : Float</td>
<td>Makes a negative number into a positive number</td>
</tr>
<tr>
<td>acos(value : Float) : Float</td>
<td>Returns the arc cosine of an angle specified in radians</td>
</tr>
<tr>
<td>asin(value : Float) : Float</td>
<td>Returns the arc sine of an angle specified in radians</td>
</tr>
<tr>
<td>atan(value : Float) : Float</td>
<td>Returns the arc tangent of an angle specified in radians</td>
</tr>
<tr>
<td>atan2(value1 : Float, value2 : Float) : Float</td>
<td>Returns the angle of the point x / y in radians</td>
</tr>
<tr>
<td>ceil(value : Float) : Int</td>
<td>Rounds a value up to the nearest integer</td>
</tr>
<tr>
<td>cos(value : Float) : Float</td>
<td>Returns the cosine of an angle specified in radians</td>
</tr>
<tr>
<td>exp(value : Float) : Float</td>
<td>Returns the base of the natural logarithm to the power of the supplied float</td>
</tr>
<tr>
<td>floor(value : Float) : Int</td>
<td>Rounds a value down to the nearest integer</td>
</tr>
<tr>
<td>isFinite(value : Float) : Bool</td>
<td>Returns true if a value is not equal to POSITIVE_INFINITY or NEGATIVE_INFINITY, otherwise false is returned</td>
</tr>
<tr>
<td>isNaN(value : Float) : Bool</td>
<td>Returns true if the passed value is ‘Not a Number’ (Nan)</td>
</tr>
<tr>
<td>log(value : Float) : Float</td>
<td>Finds the logarithm of a number</td>
</tr>
<tr>
<td>max(value1 : Float, value2 : Float) : Float</td>
<td>Returns the largest of two numbers</td>
</tr>
<tr>
<td>min(value1 : Float, value2 : Float) : Float</td>
<td>Returns the smallest of two numbers</td>
</tr>
<tr>
<td>pow(value1 : Float, value2 : Float) : Float</td>
<td>Raises the first value to the power of the second</td>
</tr>
<tr>
<td>random() : Float</td>
<td>Returns a random floating value between 0 and 1</td>
</tr>
<tr>
<td>round(value : Float) : Int</td>
<td>Rounds a value to the nearest integer</td>
</tr>
<tr>
<td>sin(value : Float) : Float</td>
<td>Returns the sine of an angle specified in radians</td>
</tr>
<tr>
<td>sqrt(value : Float) : Float</td>
<td>Returns the square root of a float</td>
</tr>
<tr>
<td>tan(value : Float) : Float</td>
<td>Returns the tangent of an angle specified in radians</td>
</tr>
</tbody>
</table>

**Divide by Zero Conundrum**

When working with numbers in haXe, you’re likely to hit a few snags that even the most seasoned developers meet from time to time. Probably the most annoying of these is the divide by zero issue. The problem is that, no matter how great computers are at crunching numbers, dividing any number whatsoever by zero produces an infinite value, and, although the Float type in haXe can deal with very
Chapter 3: Learning the Basics

large numbers, there’s no way you could fit an infinite value inside one. To help resolve this issue, the haXe Math class provides two constants that allow you to test for such a mistake: NEGATIVE_INFINITY and POSITIVE_INFINITY. These two constant values are of the type Float, but do not represent actual values. They merely allow you to compare your own Float values against them to see if they are valid numbers:

```haXe
var myFlt : Float = 2 / 0;
var isValid : Bool = (myFlt != Math.POSITIVE_INFINITY);
```

You can also use the Math method isFinite, which returns true if the contained number is not an infinite value:

```haXe
var myFlt : Float = 2 / 0;
var isValid : Bool = Math.isFinite(myFlt);
```

**Not a Number**

Another way to test for numerical validity is using the NaN constant. NaN, which means Not A Number, can be used to check if a Float or Int are valid numerical types. If a numerical type is not valid, then its value is NaN:

```haXe
var myFlt : Float;
untyped { myFlt = "Not a Number"; }
var isValid : Bool = (myFlt != Math.NaN);
```

On Neko, where such an expression is valid, the isValid variable will test true, while in Flash, the result will not be a valid number, and so will be false. The haXe Math class also provides a method called isNaN. This method returns true if the numerical value is not valid, and can be used in the following way:

```haXe
var myFlt : Float;
untyped { myFlt = "Not a Number"; }
var isValid : Bool = !Math.isNaN(myFlt);
```

As you can see, the negation operator (!) was used to return true if the result of the method is a valid numerical type.

**Float to Int Conversion Functions**

When converting a Float value to an Int, the virtual machine just simply disregards the mantissa (the value after the decimal point). This is the equivalent of rounding the Float down to the nearest integer. The haXe Math class provides some more controlled methods for converting a Float to an Int. These include floor, ceil, and round.

The floor method performs the same functionality as casting a Float to an Int, rounding the floating point value down to the nearest whole integer:

```haXe
var myInt : Int = Math.floor(myFlt);
```
Part I: The Core Language

The `ceil` method, or ceiling performs the opposite functionality by rounding a floating point value up to the nearest integer, while the remaining method, `round`, will round the value up if the mantissa is equal to or larger than .5 and down if the mantissa is less than .5 in Flash and JavaScript and will round up if the mantissa is larger than .5 and down if the mantissa is less than or equal to .5 in Neko.

Using String Functions

Three classes of functions are provided by the haXe library for manipulating Strings. The most common of the string functions are contained within the `String` class itself, while the less useful functions can be found in the `StringTools` class. The remaining class is the `StringBuf` class, which is used for creating very large strings from smaller strings.

The String Class

Many of the methods provided by the `String` class provide the means to treat strings as though they were Arrays of characters. This offers a large amount of leverage to search and replace sections of a string as you would items in an Array.

Table 3-10 provides a list of the `String` class methods.

<table>
<thead>
<tr>
<th>String Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length : Int</td>
<td>Returns the number of characters in a string; it is a read-only property. Escape characters, including forward slash, count as one character.</td>
</tr>
<tr>
<td>charAt(index : Int) : String</td>
<td>Returns the character at a given location.</td>
</tr>
<tr>
<td>charCodeAt(index : Int) : Int</td>
<td>Returns the ASCII representation of a character at a given location.</td>
</tr>
<tr>
<td>indexOf(value : String, ?startIndex : Int) : Int</td>
<td>Returns the first location past a given starting point of a given character or string.</td>
</tr>
<tr>
<td>lastIndexOf(value : String, ?startIndex : Int) : Int</td>
<td>Returns the first location past a given starting point of a given character or string whereby the string to search is traversed in reverse.</td>
</tr>
<tr>
<td>split(delimiter : String) : Array&lt;String&gt;</td>
<td>Splits a string into an Array from a given delimiter character or string.</td>
</tr>
<tr>
<td>substr(pos : Int, ?len : Int) : String</td>
<td>Returns a section of a string.</td>
</tr>
<tr>
<td>toLowerCase() : String</td>
<td>Returns the string in all lowercase characters.</td>
</tr>
<tr>
<td>toUpperCase() : String</td>
<td>Returns the string in all uppercase characters.</td>
</tr>
<tr>
<td>toString() : String</td>
<td>Returns a string representation of the string.</td>
</tr>
<tr>
<td>static fromCharCode(code : Int) : String</td>
<td>Converts the passed numeric code into an equivalent character.</td>
</tr>
</tbody>
</table>
Extracting Characters from a String

Accessing characters at a particular location of a string Array is performed using the `charAt`, `charCodeAt`, and `substr` methods. The `charAt` and `charCodeAt` methods can return a single character from a string at a given location; `charAt` would return its character representation as a `String`, while the `charCodeAt` would return its ASCII representation as an `Int`. The `substr` method accepts a second parameter, which determines the length of the string of characters to return.

```javascript
var newStr : String = str.charAt(3); // will return `e` from "haXe and Neko"
var newInt : Int = str.charCodeAt(3); // will return 101 from "haXe and Neko"
var newStr : String = str.substr(2, 4); // will return `Xe a`
```

Searching a String

In most situations, you will need to search through a string before you know what part of that string you want to return. For example, you may be parsing a configuration file with properties and their values and want to store the values into a Hash Table using the properties as a key. To search a string, you use the `indexOf` method, which requires the string or character you want to find in the string you are searching. You can also choose to provide a second parameter, which is the location you want to start your search, though it is optional. When the character or string is found in the parent string, the location is returned, while if no instance of the character or string is found, then -1 is returned:

```javascript
var location : Int = str.indexOf("e"); // will return 3 from "haXe and Neko"
var location : Int = str.indexOf("e", 4); // will return 10
var location : Int = str.indexOf("e", 11); // will return -1
```

As well as searching through a string with `indexOf`, you can also perform the same feat in reverse, starting at the end of the string, with the `lastIndexOf` method:

```javascript
var location : Int = str.lastIndexOf("e"); // will return 10 from "haXe and Neko"
```

When searching a string, it often helps to know how long the string is. You can do this by querying the string’s length property:

```javascript
var lengthOfString : Int = str.length; // will return 13 from "haXe and Neko"
```

Converting a String to an Array

Finally, if performing functions against a string as though it were an Array isn’t quite enough, you can convert a string to an actual Array of strings or characters (`Array<String>`) using the `split` method. The `split` method takes only one parameter: a delimiter. A delimiter is a character or string that represents the middle of the Array items in your string. Therefore, in a string where a comma separates the items, using the comma as the delimiter will extract the items themselves and store them in the Array:

```javascript
var items : String = "item1,item2,item3";
var itemArray : Array<String> = items.split(",");
```

You can have as long a delimiter as you like, though the delimiters are discarded in the split.
Converting an Array Back to a String

Once you perform the necessary operations against the items in your array, you can convert it back into a string using the `Arrays.join` method. `join` takes as its parameter the delimiter you want to use to glue the items together and then returns the newly joined string:

```javascript
var strArray : Array<String> = ["item1","item2","item3"];
var wholeStr : String = strArray.join(","); // will return "item1,item2,item3"
```

Here is an example of parsing a string of properties and their values into a Hash Table. Type the code into a new file and save as `StringParsing.hx`:

```javascript
class StringParsing
{
    public static function main()
    {
        var configStr : String = "item1=val1 & item2=val2";
        var keyStr : String;
        var valStr : String;
        var htProps : Hash<String> = new Hash();
        var prevKeyLoc : Int = 0;
        var curKeyLoc : Int = 0;
        var curValLoc : Int = 0;
        curKeyLoc = configStr.indexOf("=");
        curValLoc = configStr.indexOf(" & ");
        keyStr = configStr.substr(0, curKeyLoc);
        valStr = configStr.substr(curKeyLoc+1, curValLoc - curKeyLoc-1);
        htProps.set(keyStr, valStr);
        prevKeyLoc = curKeyLoc;
        curKeyLoc = configStr.indexOf("=", prevKeyLoc+1);
        keyStr = configStr.substr(curValLoc+1, curKeyLoc - curValLoc-1);
        valStr = configStr.substr(curKeyLoc+1, configStr.length - curKeyLoc-1);
        htProps.set(keyStr, valStr);
        trace(htProps.get("item1"));
        // Outputs: val1
        trace(htProps.get("item2"));
        // Outputs: val2
    }
}
```

The `main` function provides a number of temporary variables for storing the location of the most recent search. As you traverse the string contained in the `configStr` variable, you note the beginning and end of property and value pairs. You know that a property name will start the string and that property names exist after an assignment operator (`=`) and before an ampersand (`&`). You also know that the property values exist at the end of the string and between any ampersand and assignment operator. With this information, you can search for the locations of each ampersand and assignment operator and extract the information between them for use in your Hash object.
Chapter 3: Learning the Basics

The StringTools Class

The StringTools class provides a number of simple methods used for tidying your strings or converting them into a safe format. It may be that you need to remove erroneous spaces from the beginning or end of a string submitted by the user of your application. Maybe you are transferring them across the Internet and wish to prepare the string so it is not mangled in the delivery process or you are simply storing the string in a file or database and want to prepare first. Whatever it is, StringTools can help.

Table 3-11 details the methods of the StringTools class.

Table 3-11

<table>
<thead>
<tr>
<th>StringTools Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseDecode(s : String, base : String) : String</td>
<td>Returns a decoded version of an encoded string from a given base (must be power of 2)</td>
</tr>
<tr>
<td>baseEncode(s : String, base : String) : String</td>
<td>Returns an encoded version of a string from a given base (must be power of 2)</td>
</tr>
<tr>
<td>endsWith(s : String, end : String) : Bool</td>
<td>Returns true if a string ends with the given character or string</td>
</tr>
<tr>
<td>hex(n : Int, ?digits : Int) : String</td>
<td>Encodes a number into a hexadecimal representation, with an optional number of zeros for left padding</td>
</tr>
<tr>
<td>htmlEscape(s : String) : String</td>
<td>Returns a copy of the string with all necessary symbols escaped for display in an HTML browser</td>
</tr>
<tr>
<td>htmlUnescape(s : String) : String</td>
<td>Returns an unescaped copy of an htmlEscaped string</td>
</tr>
<tr>
<td>isSpace(s : String, pos : Int) : Bool</td>
<td>Returns true if a given location in a string is a space character</td>
</tr>
<tr>
<td>lpad(s : String, c : String, l : Int) : String</td>
<td>Returns a copy of a string with the left-hand side padded with a given character or string a given number of times</td>
</tr>
<tr>
<td>ltrim(s : String) : String</td>
<td>Returns a copy of a string with all spaces on the left of the string removed</td>
</tr>
<tr>
<td>replace(s : String, sub : String, by : String) : String</td>
<td>Returns a copy of a string with all occurrences of a given character or string replaced for another given character or string</td>
</tr>
<tr>
<td>rpad(s : String, c : String, l : Int) : String</td>
<td>Returns a copy of a string with the right-hand side padded with a given character or string a given number of times</td>
</tr>
<tr>
<td>rtrim(s : String) : String</td>
<td>Returns a copy of a string with all spaces on the right of the string removed</td>
</tr>
</tbody>
</table>

Table continued on following page
Part I: The Core Language

<table>
<thead>
<tr>
<th>StringTools Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>startsWith(s : String, start : String) : Bool</td>
<td>Returns true if a string starts with the given character or string</td>
</tr>
<tr>
<td>trim(s : String) : String</td>
<td>Returns a copy of a string with all spaces on both ends of the string removed</td>
</tr>
<tr>
<td>urlDecode(s : String) : String</td>
<td>Returns a decoded copy of a URL-encoded string</td>
</tr>
<tr>
<td>urlEncode(s : String) : String</td>
<td>Returns a copy of a string encoded for transport within a URL</td>
</tr>
</tbody>
</table>

Dealing with Unwanted Spaces

It is common for strings retrieved from the user to contain unwanted spaces. The most ugly of these are those appended to the beginning or end of your string, which can throw your string algorithms if unexpected. Luckily, StringTools provides several methods for dealing with these pesky characters.

You can check if a character in your string is a space character using the isSpace method. This method accepts the string you wish to examine as the first parameter and the location of the wanted or unwanted space character as the second parameter. If the character at that location is indeed a space, then the Boolean value true is returned:

```java
var tmp = StringTools.isSpace(myStr, 4); // returns true from "haXe and Neko"
```

Removing unsightly spaces from the beginning or end of your string can be performed using trim, ltrim, and rtrim. The trim method removes spaces from both sides of your string, while ltrim only removes spaces from the left side of your string and rtrim only from the right. All three of these methods will remove every space that exists at the relevant end of your string, even if it has a bazillion of them.

```java
StringTools.trim(myStr);
```

More String Manipulation

You can verify the starting and ending characters of a string using the startsWith and endsWith methods. Both take the string to examine as the first parameter and a string of one or more characters you wish to compare as the second parameter:

```java
var isValid : Bool = StringTools.startsWith(myStr, "haXe");
// returns true for "haXe and Neko"
isValid = StringTools.endsWith(myStr, "rocks");
// returns false
```

Sometimes, you might find it necessary to add characters to the ends of your string, so that they might fit into a field in a database for example. You can do this using the lpad and rpad methods. Both methods require the string to pad, the string you want to append as padding, and the number of characters the overall string should be:

```java
var paddedString : String = StringTools.lpad(myStr, " ", 20);
```
Finally, you can replace all occurrences of a particular character or string of characters using the `replace` method. This is helpful when exchanging a delimiter in your string for a different character:

```haxe
var oldStr : String = "item1,item2,item3";
var newStr : String = StringTools.replace(oldStr, ",", ";");
```

### Encoding Strings for the Web

The `StringTools` class provides several methods for safeguarding your strings from the perils of transferring them across the Internet. If you are transferring your strings using the `GET` method, you will need to encode them correctly so that the string is sent without being modified on its journey and to help guarantee that it makes it in one piece. You can do this in haXe using the `urlEncode` method:

```haxe
var encodedStr : String = StringTools.urlEncode("String to send ");
```

Retrieving an encoded string on the other end of its journey can then be accomplished using the `urlDecode` method:

```haxe
var newStr : String = StringTools.urlDecode(receivedStr);
```

If you then want to display that string in a browser, some of the symbols you need to display may be read differently as HTML symbols, so will need to be escaped. You can do this using the `htmlEscape` method:

```haxe
var escapedStr : String = StringTools.htmlEscape("String to display");
```

Again, you can reverse the effects of this method using the `htmlUnescape` method:

```haxe
var normalStr : String = StringTools.htmlUnescape(escapedStr);
```

### The StringBuf Class

The `StringBuf` class provides the developer with a clean way of constructing strings from much smaller strings and characters.

Table 3-12 details the methods contained in the `StringBuf` class.

### Table 3-12

<table>
<thead>
<tr>
<th>StringBuf Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(?x : Dynamic) : Void</code></td>
<td>Adds a string representation of an object to the <code>StringBuf</code> string</td>
</tr>
<tr>
<td><code>addChar(c : Int) : Void</code></td>
<td>Adds a character from an ASCII code to the <code>StringBuf</code> string</td>
</tr>
<tr>
<td><code>addSub(s : String, pos : Int, ?len : Int) : Void</code></td>
<td>Adds a section of a string from a given location and length to the <code>StringBuf</code> string</td>
</tr>
<tr>
<td><code>toString() : String</code></td>
<td>Returns the string contained in the <code>StringBuf</code> object</td>
</tr>
</tbody>
</table>
Part I: The Core Language

The `StringBuf` class must be instantiated before you can use it:

```plaintext
var myString : StringBuf = new StringBuf();
```

Once the `StringBuf` class is instantiated, you can then append your smaller strings to the `StringBuf` object using the `add` method. The `add` method can take any type of object you choose, and will extract a string representation from that object if it exposes one:

```plaintext
myString.add(someObject);
myString.add("more text");
```

If it is a part of a string that you want to add the `StringBuf` object, you can use the `addSub` method. This method works in exactly the same way as the `String` class’s `substr` method:

```plaintext
myString.addSub("Big long text string ", startPos, endPos);
```

You can even add ASCII character codes that will then be converted to the character representations before being appended to your `StringBuf` object. You do this using the `addChar` method:

```plaintext
myString.addChar(101); // adds the character `e` to the StringBuf object
```

Finally, you can extract the string back out of the `StringBuf` object using the `toString` method:

```plaintext
var str : String = myString.toString();
```

Summary

Wow, this was been quite a chapter and much was discussed. Throughout the chapter, you learned the following skills and information:

- The various types in the haXe language
- Using `untyped` code in haXe
- Using the `Date` object
- Using `Arrays`, `Lists`, `Hash`'s, and `IntHash` objects
- Manipulating values with operators
- Manipulating numerical values with the `Math` class
- Manipulating strings with the `String`, `StringTools`, and `StringBuf` classes

The next chapter introduces functions, conditional statements, loops, and iterators. Some of the information in the next chapter reevaluates areas you have already covered, but all of it will be new and challenging.
Controlling the Flow of Information

Moving on from types and data manipulation brings you to how your data will flow in your applications. haXe supports a very rich array of functionality, which has been borrowed from the ML family of functional languages and tightly wrapped around the now ubiquitous object-oriented coding conventions. You will look at some of these practices in this chapter, including:

- Conditional statements
- Loops
- Iterators and collections in loops
- Functions and local functions
- The haXe Lambda class

Pushing Data Around

It’s all very well having nicely labeled cubby holes to put your data, but you’re likely to want to apply various processes to that data so that it facilitates some form of utility. Doubtlessly, you will already be able to implement certain functionality into your haXe applications utilizing your own experiences with other object-oriented languages, as many languages following the European Computer Manufacturers Association (ECMA) standard provide a very similar pattern of employment. However, as the haXe language author, Nicolas Cannasse, is so very fond of the functional ML family of languages, numerous variances have been added to haXe that provide a surprising amount of powerful extended features. For this reason, even the most seasoned developer is advised to read this chapter thoroughly, before boldly charging into haXe development.
Part I: The Core Language

The functional programming capabilities offered by haXe provide the facility to apply cleaner, smaller, and faster processes on data in the right circumstance. While the flow of data can be channeled in a fashion that may be considered traditional for object-oriented programming, haXe provides the means to break out of these rigid structures and enables your data to take on an almost fourth dimensional relationship with your classes. In haXe, object-oriented practices are not king, but merely a means to an end. While classes are a necessity at the very root of your haXe applications, the overall perspective and direction of the haXe framework offer the use of a more patterned methodology that can free your data of the confines stressed by object-oriented Programming (OOP) attitudes and provide a way for your data to be managed and affected in a more linear manner. When considering object orientation and how it applies to applications development, one often attempts to attain a balance between the structured containment of objects over data with a level of abstraction and reusability. However, by its very nature, the structural definition applied to data in object orientation renders the possibility of absolute abstraction as an almost impossible task. haXe helps to alleviate these issues a little by providing the best of both worlds: the organizational facilities of object-oriented design with the applied abstract expressions of functional programming.

Choosing a methodology in haXe is not necessary. The combined features of both programming methodologies form to provide an almost organic experience to make developing with haXe feel almost as though you were sculpting your application from clay. Almost certainly, the way you construct your applications with haXe will focus more closely upon the style of programming with which you are most familiar, but know you can be left at no disadvantage when applying the scales to the levels of which both styles have been employed.

Conditional Statements

Depending on your perspective, conditional statements can be seen as one of the most basic forms of data flow processes, as they perform the simple task of herding your data through your application logic based on decisions. These decisions can be achieved using conditional expressions (expressions evaluated through the use of conditional operators) or as the return value of a function or calculated expression. In haXe, a condition is a Boolean value of either true or false and can be represented by no other type. If a decision is to be made on an expression returning a type other than a Boolean, then the Std.bool() function can be applied to cast the expression into a Boolean value. For example:

```haXe
// produces a compile error
if ( 3 + 2 ) someVar = someValue;
// will compile successfully
if ( Std.bool( 3 + 2 ) ) someVar = someValue;
```

if Statements

The if statement is the most basic and malleable of the conditional keywords and will likely supply the greater number of conditional statements in your applications. The conditional expression is supplied in brackets after the if keyword. If the expression returns true, then the statement of code following the expression is executed:

```haXe
if ( 3 == 3 )
    someVar = someValue;
// rest of application
```
Chapter 4: Controlling the Flow of Information

If you have more than one statement you want executed following a true expression, you can form a block by surrounding those lines of code with curly braces { and }:

```java
if ( 3 == 3 )
{
    someVar = 1;
    someVar++;
}
// rest of application
```

The line or block of code following an if statement is only executed if the statement returns true. You can reverse the return value of a conditional expression by supplying an opposing conditional operator, or by negating the entire expression with the negation operator (!):

```java
if ( 3 != 3 )
// do this...
```

The preceding code is the same as:

```java
if ( ! 3 == 3 )
// do this...
```

The negation operator alters the return value of a conditional expression so that true will be returned instead of false, and false returned instead of true. It does not change the if statement rules, so true must still be returned from the expression for the proceeding code to be executed. If, however, you require code to be executed following a false conditional expression, you can use the else keyword, which provides an alternative route of execution in your logic. The else keyword must immediately follow the code or block executed by an if statement:

```java
if ( 3 == 3 )
// if true, execute this line
else
// otherwise, execute this line
```

It is possible to nest if statements so that multiple conditions can be met before executing certain lines of code:

```java
if ( thatVar == thatValue )
{
    if ( thisVar == thisValue )
    {
        // do code
    }
}
```

Nesting if statements mean that the nested statement will only evaluate if the outer statement is true. You can combine the expressions in the if statements to form a single expression using the conditional operators. Therefore, you could rewrite the example as:

```java
if ( thatVar == thatValue && thisVar == thisValue )
{
    // do code
}
```
When you combine expressions in this way, the expressions are still considered separate by the virtual machine in terms of execution. The meaning is, if the first of the two expressions returns false, then the second expression will not be evaluated, so any data modifications performed in the second expression will not happen. For example:

```javascript
var myInt = 3;
if ( 4 == 3 & & ++myInt == 4 )
{
   // do code
}
trace( myInt );
```

Here, one would expect `myInt` to equate to the value 4 as it is pre-incremented. However, because the expression `4 == 3` returned `false`, the second part of the expression was never evaluated, so `myInt` is traced as the value 3.

Nesting `if` statements are often preferential, though there are times when it is a necessity because of logic flow. However, by combining conditional expressions into a single `if` statement, there will be a slight performance increase as only one expression is validated, not two. However, this increase is negligible, and so should not affect your decision.

When nesting an `if` statement in an `else` block, you are in effect supplying a new branch for testing an expression’s equality. Continual nesting of `if` statements in the parents `else` clause will thereby provide a chain of equality expressions offering numerous chances for an expression to return `true`:

```javascript
if ( myVar == 1 )
   // do this...
else
   if ( myVar == 2 )
      // do this
else
   if ( myVar == 3 )
      // repeat as necessary
...
else
   // if all else fails, do this
```

`if` statements used in this way can nest as deep as you like, though it is not necessary to indent them in this fashion. The majority of developers will often provide the nesting `if` keyword directly after the `else` keyword, providing a seemingly new statement type:

```javascript
if ( myVar == 1 )
   // do this...
else if ( myVar == 2 )
   // do this...
else if ( myVar == 3 )
   // repeat as necessary
...
else
   // if all else fails, do this
```
Chapter 4: Controlling the Flow of Information

As you can see, this will make your code look a lot neater and easier to read. The final else block used at the end of the nested if statements forms the if all else fails safety net, meaning that any comparison not yet met will be processed here.

Aligning if statements can be increased further still when using short lines of code, though this is at your discretion. For example:

```java
if ( 3 == 3 )
    myVar++;
else
    myVar--;
```

Can be more readable as:

```java
if ( 3 == 3 ) myVar++ else myVar--;
```

If you look at the last example, you might notice that the expression preceding the else keyword is not immediately proceeded by a semicolon (;) ending the line. This is to help facilitate the readability of the whole if statement and is allowed only with single lines that directly follow the conditional expression of the if statement.

### switch Statements

switch statements provide an alternative to if statements that repeatedly compare the equality of a single value. For example, you may be using an integer flag to store a particular state then perform different tasks based on that flag. If the value of that flag can be anything from 0 to 255, for example, then you may require an awful lot of elseif statements. The switch statement can help to tidy such code by making it more readable.

The switch statement is written similarly to an if statement, with the exception that the value to compare is supplied to the brackets following the switch keyword, while the various values supplied for the comparison are provided by the case keyword and are listed within the switch's code block, like this:

```java
switch ( myValue )
{
    case val1:
        // do code
    case val2:
        // do code
    ...
}
```

You can provide as many case statements as is needed for each elseif statement that would otherwise facilitate the comparison. Each case keyword is supplied the value for the comparison and the line is ended with a colon (:), which indicates the end of the case and the start of the code block to execute.
Part I: The Core Language

following a match. If you then require a final else statement to provide the catchall clause, you use the default keyword:

```
switch ( myValue )
{
   case val1:
      // do code
   case val2:
      // do code
   ...
   default:
      // if all else fails, do this
}
```

When a switch statement is executed, the virtual machine executing the code will compare the given value with each case until a match is made or the end of the switch statement has been found. In many languages, once a match is made and the relevant code for that case is executed, the next case’s code segment is executed even if the case’s comparison value is not a match, and then the next, and so on until the end of the switch block has been reached or a break keyword is found. In haXe, however, this is not the case (excuse the pun). Only the code associated with a case that has a matching comparison value will be executed. Once this has happened, the switch statement is automatically exited and the rest of the application is resumed. No break keyword is required by the case to allow this, and indeed, supplying a break in a switch statement will cause a Break in loop compiler error, as the break keyword is not associated with the switch or case keywords. The benefit to this, however, is that the default case does not have to exist at the end of a switch case list, but can be put anywhere in this list.

If you require blocks of code to be executed for several different case statements, it is advisable to supply a function as the executable code so as to facilitate maintaining the logic in one place.

Returning Values from Conditional Statements

Unlike many languages, the if and switch statements in haXe can return a value. This is extremely useful for reducing unnecessary code and increasing readability. For example, instead of performing a repetitive task of assigning a variable or calling a function in every single case and elseif block you can opt to assign the return value of the if or switch to a single representation of the variable or function. So, in the case of an if statement, you could do the following:

```
var myInt = if ( myVal == "one" )
   1;
else if ( myVal == "two" )
   2;
else if ...
```

While a switch statement may look like this:

```
var myInt = switch ( myVal )
{
   case "one":
      1;
   case "two":
      2;
}
```
Chapter 4: Controlling the Flow of Information

These conditional statements can also form the value passed to a function, so the previous examples could be output like this:

```javascript
trace( if ( myVal == "one" )
    1;
else if ( myVal == "two" )
    2;
else
    -1;
);
```

And:

```javascript
trace( switch ( myVal )
    {
    case "one":
    1;
    case "two":
    2;
    });
```

Both of these are legal! However, for the sake of readability, it is better to pass the return value to a temporary variable first, and then supply that variable to the function in question.

Returning a value from an `if` statement printed on a single line facilitates the same functionality as that of the ternary operators in the C language:

```javascript
var myInt = if ( myVal == "one" ) 1 else 2;
```

Which is the same as the C equivalent:

```c
int myInt = ( strcmp( myVal, "one" ) ) ? 1 : 2;
```

When returning a value from a `switch` or `if` statement, it is important to make sure that the type of the values being returned are matched, otherwise a compiler error will ensue. If no value is meant to be returned, then the values that may return by accident, such as those supplied indirectly by a function call, may be of differing types. If no value is returned within an `if` or `switch` statement, then the type `Void` is returned.

**Loops**

Computers are very linear machines, and a processor, with some exceptions, can only perform one instruction at any one time. Sure, your machine may look like it’s doing an awful lot at once, but in reality it is merely performing processes on one area of information, known as a thread, at a time. At some point, this thread will yield the execution spotlight to the next thread so it can do its stuff, then that one will yield to the next thread, and so on. All these instructions are processed so fast, that the machine gives the appearance that these processes are happening simultaneously, even though they’re not.
Part I: The Core Language

The same can be said of your applications. When you wish to perform a number of processes in your application on groups of data, the application must perform these processes on each item of data, one at a time, not on the group as a whole. To do this, the application iterates over each item so that as one item has finished being processed, the code will move to the next item in the group and re-perform the same logic. The iteration will then usually cease once all items have been processed.

Iteration is performed with the help of loops. The loops provide a way to repeat a chunk of code a certain number of times, though usually with differing values each time. These values will very often pertain to different objects in memory or represent flags that will keep changing until a certain value is matched and the loop can discontinue.

**The while Loop**

while loops execute while a certain expression is true. Like the if statement, the supplied expression must return a Boolean type value, though this can be forced with the use of `Std.bool()`.

There are typically two types of while loops: those with a condition that evaluates before any further loop code can be executed and those with a condition that evaluates after the loop code can be executed. The latter is known as a do while loop, as the do keyword is required:

```javascript
// while loop
while ( someVar == someValue )
{
    // do code
}
// do while loop
do
{
    // do code
}
while ( someVar == someValue );
```

As with the if statement, the block braces are not required if only one line of code is present in the loop.

The differences between the while and do while loops are that, with the while loop, the loop is only performed if the evaluated expression returns true. However, with the do while loop, the loop code will occur at least once regardless of the expression, but will only repeat if the evaluated expression returns true.

```javascript
class WhileLoops
{
    public static function main()
    {
        var myInt : Int = 0;
        // will print to screen ten times
        while ( myInt < 10 )
            trace( "displayed " + { ++myInt } + " times" );
        // will skip as myInt == 10
        while ( myInt < 10 )
            trace( "displayed " + { ++myInt } + " times" );
    }
}
```
Chapter 4: Controlling the Flow of Information

As you can see in the example, the line executed in the do while loop does not end with a semicolon (;). Like if statements, a do while loop doesn’t require a single line of code to end with a semicolon. In fact, the compiler is insistent that one does not exist. You will still need to apply them to blocks of code, however.

The for Loop

The for loop in haXe is very different from the for loop used in C and Java, and more closely resembles the common foreach loop. That is not to say that it is a foreach loop, as it clearly isn’t, but it is similar insofar as the haXe for loop iterates over objects and values rather than looping until a given expression returns true.

The haXe for loop is a loop that is directly associated with an iterator object. Iterators will be explained later in this chapter, and then more deeply in Chapter 5, “Delving Into Object-Oriented Programming,” but for the purposes of this explanation, you’ll examine the default iterator: the IntIter object. Before doing that, though, it’s probably best to explain what an iterator is. An iterator is an object that represents an iterable value. This can be a list of objects or values such as those contained in an Array, a numerical value ranging from a given minimum value to a given maximum value, or even the letters of the alphabet. The iterator will supply the values with the help of two methods that it exposes: hasNext() and next(). The hasNext() method takes no arguments and will return true if it has more objects to iterate over and false if it doesn’t. The next() method also takes no arguments and returns the next object in the iterators stack.

Anyway, many of the types and processes in haXe that make use of iterators will automatically access the hasNext() and next() methods of an iterator. The for loop is one of these.

The IntIter Object

As it has been said before, IntIter is the most basic available iterator and is the most commonly used in haXe applications, as it aids in iterating from one given value of type Int to another given value of type Int. You can use the IntIter object by first instantiating it as you would an Array while supplying it with your minimum and maximum values, which are assigned to hidden variables labeled min and max respectively. You then pass a variable identifier and the instanced object to the for loop for iteration and separate them with the in keyword, like this:

```haXe
var iter = new IntIter(0,10);
for ( i in iter )
    trace ( i );
```
Part I: The Core Language

When the loop is run, the following procedure is carried out:

1. The for loop checks that the iterator object has values to iterate by making a call to the iterator’s hasNext() method. This method evaluates whether \( \text{min} \) is less than \( \text{max} \) (\( \text{min} < \text{max} \)) and returns the result of this expression. If it returns true, then there is at least one value to iterate, otherwise the for loop exits.

2. The in keyword assigns the return value of the iterator’s next() method to the variable provided on the left-hand side of the in keyword. This assignment is performed in the same way as the expression \( \text{myInt} = \text{min} \), which means that, if the value to assign is not an object, then a copy is assigned, otherwise a reference is assigned. In the case of an \( \text{IntIter} \), an Int is always assigned.

3. After the value of \( \text{min} \) is assigned, it is incremented by 1 ready for the next loop. Once the code supplied to the for loop has been executed, the process repeats from step 1.

As the value of \( \text{min} \) is evaluated as being less than \( \text{max} \), the range of numbers supplied in the loop will never be equal to the \( \text{max} \) variables value. Therefore, in your example, the loop would write the values 0 to 9 on the screen.

*When supplying the variable to the left-hand side of the in keyword in a for loop statement, you do not include the var declarator, as the variable will be declared for you.*

**The IntIter Operator**

The \( \text{IntIter} \) iterator is quite special when compared to other iterator types, as one can be declared and instantiated automatically using the iterator operator (\.\.\.). Using this operator, you can rewrite the previous example like this:

```haXe
for ( i in 0...10 )
    trace( i );
```

The \( \text{IntIter} \) iterator cannot operate in a decrementing fashion, so counting backward from 10 to 0 is not possible. However, you can imitate this process using a trick, by deducting one more than the value of the current iteration from the maximum value:

```haXe
for ( i in 0...10 )
    trace( 10 - ( i + 1 ) );
```

**Looping Over Collections**

Most applications that use Arrays and Lists will likely want to loop through the contained values at some point. In fact, the List has been highly customized to this end and would be pretty useless without some form of iteration being applied to its content. To facilitate this, each of the collection objects in haXe supplies a method called iterator(). This method returns a copy of an iterable object containing the items of the collection that can be passed to processes such as a for loop:

```haXe
var myArray : Array<String> = ["item1", "item2", "item3"];
var arrIter = myArray.iterator();
for ( i in arrIter )
```
Chapter 4: Controlling the Flow of Information

```javascript
{
    trace( i );
}
```

This example displays each of the items in the Array. However, there is an easier way to do this. As a convenience, the `for` loop happens to be able to extract the iterator object directly from any object exposing a method `iterator()` with a return type `Iterator<T>`. Here, the `<T>` is a placeholder and is used to specify the type of value held in the iterator, so in this example, the return type of `iterator()` will be `Iterator<String>`:

```javascript
var myArray = ['item1', 'item2', 'item3'];
for ( i in myArray )
{
    trace( i );
}
```

**Direct Accessing of Collection Items**

The values stored in the iterator object will pass copies of the values contained in any relative collection if the type of the value is not an object; otherwise a reference is passed. This can be a little problematic if you want to modify the values stored in the Array of, say, strings but want those modifications to perpetuate in the Array once the loop has ended. Take a look at the next example:

```javascript
var myArray = ['item1', 'item2', 'item3'];
for ( i in myArray )
{
    i += '0';
    trace( i );
}
trace( myArray[0] );
// Outputs: item1
```

When the values are written to screen, you can see that each of the items in the Array has been appended with the character 0, yet when the first item of the Array is queried outside of the `for` loop, it appears without the appended character. Sometimes, this may very well be the effect you wanted, but what if it’s not?

Under such circumstances, you can abstract the use of an iterator by using the `IntIter` object and setting its maximum value to the length of the Array. This way, you can directly access the Array items and guarantee that the items are modified. Using this method, your previous example will look like this:

```javascript
var myArray = ['item1', 'item2', 'item3'];
for ( i in 0...myArray.length )
{
    myArray[i] += '0';
    trace( myArray[i] );
}
trace( myArray[0] );
// Outputs: item10
```

On running the example, the character appended to the first item should be notably permanent, or at least until the application ends.
Part I: The Core Language

Iterating Over Hash Tables

The Hash and IntHash objects expose an iterator() method much like the Array and List objects. However, only the values of the contained items are passed to the iterator. As the indexes of these objects are accessed through the use of keys, it is not possible to use the IntIter object to directly access the values they contain, unless the object is an IntHash with sequential key values, which really defeats the object of these types. To get around this issue, an instance of either type provides a keys() method, which returns an iterator containing the keys used in the collection. Once attained, the iterator can then be used in place of the IntIter object for direct access to the collections items:

```javascript
var myHash = new Hash<String>();
myHash.set( "one", "doh" );
myHash.set( "two", "ray" );
myHash.set( "three", "me" );
for ( i in myHash.keys() )
{
    myHash.set( i, myHash.get( i ) + "#" );
    trace( myHash.get( i ) );
}
trace( myHash.get( "one" ) );
// Outputs: doh#
```

Break and Continue

There are two special keywords for use with both the while and for loops. These are break and continue.

You use break when you want to exit a loop and continue with the rest of your application. For example, when using a for loop, you may decide to include an extra contingency that will cause the loop to end for the benefit of certain business logic or to avoid possible errors:

```javascript
var myArray1 = [1,2,3,4];
var myArray2 = [10,9,8];
for ( i in 0...myArray1.length )
{
    if ( myArray2.length <= i ) break;
    trace( myArray1[i] + " " + myArray2[i] );
}
```

Here, the length of myArray2 is compared with the iterated value to guarantee that a value exists in the Array for display, as only one Array can be used to initiate the loop. If, at any time, the iterated value exceeds the number of items in the second Array, the loop is forced to exit and the application continues as normal.

The continue keyword is used when you wish to skip a single iteration in a loop. For example, if you wanted to iterate an integer and display any values that are not a multiple of 3, you could do the following:

```javascript
for ( i in 0...10 )
{
    if ( i % 3 == 0 )
    {
```
Chapter 4: Controlling the Flow of Information

```
continue;
trace(i); // Outputs: 1, 2, 4, 5, 7, 8
```

**Functions**

Functions perform the task of grouping code that is used frequently; that is to say, used more than once. In haXe, functions provide a much more rich entity than many other object-oriented languages thanks to the inclusion of some functional behaviors adopted from ML languages. You will be looking at the various uses of functions in this chapter, including their more dynamic nature and how they apply to some of the types that exist in the haXe framework.

In haXe, there are two types of functions: those used as methods of a class and those specified as a variable type, known as local functions. The class methods can be further grouped as static functions or instance methods, though these variations are discussed in Chapter 5, “Delving Into Object-Oriented Programming.”

You’ve already seen one of these function types in use in the form of the static main function. This function performs the objective of entry point into your application, so it is always the first function called. What is then called or created is the responsibility of that function. If the control of the application is passed on to a new function or object, the flow of control will eventually return to the static main function at some point before the application is ended. Therefore, you can originate both the necessary starting functionality and closing functionality from this one location and guarantee that both are called except in situations involving a system or application failure.

**Class Functions**

Class functions are functions that are prototyped in a class. Each function in your application will perform a specific task, such as parsing an e-mail address or reading data from a configuration file. As you learn in the next chapter, those functions that exist as class functions will usually perform processes that are of some relation to the parent class. All class functions take the following structure:

```haXe
[public/private] [static] function <name>( param1 : paramType [, param2 : paramType], ... ) : returnType
{
...
}
```

The public, private, and static identifiers are discussed in the next chapter, so you’ll skip those for now. Next is the `function` keyword, which states that what follows is the declaration of a function. The name of the function comes next and is restricted by the same rules as haXe variables, which states that it can consist of underscores (_), letters, and numbers, but must not start with a number and must not be the same as a haXe keyword. Under the general haXe naming conventions, function names always start with a lowercase letter, though this is at your discretion.
Part I: The Core Language

Following the function name are the parameter declarations, which are surrounded by round brackets. These parameters are declared similarly to local variables without the var keyword and follow the same type inference rules. You can have as many parameters in a function as you like, though less is best if at all possible, for ease of use. When your function is used, values will be passed with the function call that represent the parameters supplied in the function prototype and will be required to be of the same type or to extend the type of the parameter. The function can then use these values by accessing them with the parameter name as it would any variable.

The last part of the function is the return value. Return values are the value of the data that is returned from the function once the necessary processes are complete. If no value is returned, the type specified must be Void.

Here’s a simple function in action:

```plaintext
public static function main()
{
    var i : Int = 0;
    for ( j in 1...10 )
    {
        i = add( i, j );
        trace( i );
    }

    public static function add( num : Int, numToAdd : Int ) : Int
    {
        num += numToAdd;
        return num;
    }
}
```

Here, the simple function `add` adds the value `numToAdd` to the value `num`. When the loop is first iterated, the value of `j`, which is 1, is added to the variable `i`. On subsequent iterations, the value of `j` is incremented by one before being added to the variable `i`, so the value `i` contains with each iteration is that of a triangle number. If you run the example, you should be presented with the values 1, 3, 6, 10, 15, 21, 28, 36, and 45.

**Returning from a Function**

The return keyword allows a function to end anywhere in its body code. For example, if you wanted to exit a function early, you could place the return keyword where you wanted the function to exit, and the rest of the function code would be skipped:

```plaintext
public static function someFunction()
{
    // do code
    if ( i < 20 )
    {
        return;
    }
    // more code
}
```

Here, the pretend function `someFunction` would exit if the variable `i` is less than the value 20.
Chapter 4: Controlling the Flow of Information

Using the `return` keyword, you could also have the function return a value by supplying the value to return after the `return` keyword:

```java
public static function someFunction() : Int
{
  // do code
  if ( i < 20 )
    return i;
  // more code
}
```

Here, the value of `i` is returned from the function when the function is exited.

*Functions that do not return a value are said to return Void.*

### Function Type Inference

The types assigned to each parameter of a function do not need to be specified but can be set by the compiler using type inference. This means that parameters that are type inferred won’t have a type until a value has been passed to the function for this parameter. Once passed, the parameter is then typed with the values type from then onward.

This can cause a slight issue depending on the type of the value first sent to the function. For example:

```java
public static function main()
{
  display( 2 );
  display( 2.5 );
}
public static function display( num )
{
  trace( num );
}
```

Apart from being fairly useless, the `display` function poses a problem. You know that the type `Int` is an extension of the type `Float`, so a parameter that is typed as a `Float` should be able to accept either type, but if you first pass an `Int` type to the function for the parameter, the parameter will then be typed as an `Int` so `Floats` will no longer be accepted. The only secure way around this issue is to append the type `Float` to the `num` parameters prototype rather than leave the typing to type inference.

### Dynamic Functions

Using the `Dynamic` type comes into its own when used with functions. There are many processes you can apply to values that are relevant to numerous data types. For example, the addition operator (+) can be used with `Floats`, `Ints` and `Strings`. With `Floats` and `Ints`, the addition operator adds the values together, while using it with a pair of `Strings` will concatenate them. This means that the following example is perfectly legal and useful, if only slightly:

```java
public static function main()
{
  // Outputs: 4
  trace( double( 2 ) );
}```
Part I: The Core Language

(continued)

```java
public static function double( num : Dynamic ) : Dynamic
{
    return num + num;
}

public static function main()
{
    // Outputs: 4
    trace( double( 2 ) );
    // Outputs: 5
    trace( double( 2.5 ) );
    // Outputs: haXe haXe
    trace( double( "haXe" ) );
}
```

When you use the `Std.is()` function, it should be possible to provide alternative functionality for types where certain processes aren’t quite suitable. For example, you could rewrite the previous example like this:

```java
public static function double( val : Dynamic ) : Dynamic
{
    if ( Std.is( val, Float ) || Std.is( val, Int ) )
        return val + val;
    else if ( Std.is( val, String ) )
        return val + " " + val;
    else if ( Std.is( val, Array ) )
        return val.concat( val );
    else
        return null;
}
```

**Recursive Functions**

A recursive function is a function that calls itself. In doing so, the function creates a loop, similar to the `while` and `for` loops, and requires that certain criteria be set in order to end the recursion. The benefits of recursive functions are that you ultimately have more control over the looping in your code and require only one level of functionality rather than two involved in having a loop nested in a function.

The best benefit by far with recursion, however, is that the function calls return values for each iteration. This is a little difficult to explain without some kind of visual aid, so it is illustrated in Figure 4-1.

88
As you can see, with normal loops, you start off with a set of values, and then as each loop occurs, the values are modified in some way that creates a slightly different outcome from the code in the body of the loop. You can mimic this with recursive functions, also, but the real power comes from the values they return. Each call to the function is handled by a previous call of the function, so the calls are, in effect, nested. While the recursion takes place, the point of execution buries itself into each new function call. Then, when the recursion eventually ends, the point of execution comes back out of each function call returning a value with it. This forms the effect of looping in one direction, then reversing the loop the next.

Let’s see this in a couple of examples. First you can mimic the effect of a while loop with the following:

```plaintext
public static function main()
{
    var i = 0;
    i = loop( i );
    trace( i ); // Outputs: 20
}
public static function loop( num : Int ) : Int
{
    return if ( num < 20 )
    {
        // do code
        loop( num + 1 );
    } else num;
}
```

When executed, the `loop` function performs in a similar way to a `while` loop with the exception that the `loop` function returns the value representing the number of iterations that have taken place. The `loop` function also passes the return value of each recursive pass back to the `loop` function, though this is not
Part I: The Core Language

as evident in this particular example. To see this in action, it is better to provide code to track the recursion before and after the recursive function call:

```javascript
public static function main()
{
    var i = 0;
    i = loop(i);
    trace(i); // Outputs: 0
}
public static function loop( num : Int ) : Int
{
    var ret = 0;
    return if ( num < 6 )
    {
        trace(num);
        ret = loop(num + 1);
        trace(ret);
        --ret;
    } else num;
}
```

In this example, the value first passed to the `loop` function is incremented with each call to the `loop` function during the recursion. As the recursive calls are made in the middle of the `loop` function, the execution of each call ends at that point while the new call is being made. Eventually, the incrementing value reaches the value 6, so the `loop` function is no longer recursively called, and instead, the last passed value is returned. This then has a knock on effect for each of the parent `loop` calls, which in turn decrement the value before passing it back. The result is the display of the values 0, 1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1, 0.

**Local Functions**

Local functions are functions that exist as the value of a variable. That is to say, you construct a function and pass a reference of that construction to a variable. C developers might consider this the equivalent of function pointers, and indeed, local functions can be used to provide similar functionality. The differences between methods of a class and local functions are structural, as local functions do not have a name. However, to some extent, both functions are interchangeable, and you’ll look at ways of doing this a little later.

Local functions take on the following structure:

```javascript
var functionVariable = function( param : ParamType [, ...] ) : ReturnType
{
    // function code...
}
```

As you can see, the brackets containing the parameters directly follow the function keyword and no name is given. The prototype of the function is directly assigned to a variable by reference. Once assigned, it is then possible to copy the reference to a new variable. However, the structure of local variables also contain a type structure that must remain consistent in any new variables that the function is assigned to.
Local Function Type Structure

When declaring a variable to contain a local function, the type of that variable can be assigned through type inference, or the type can be assigned at design time. Typing a variable for local functions follows a very functional language approach where alerting the compiler that a function is expected is not enough. Instead, each of the parameter types, and the function’s return type, must be specified so that any exchange of function reference assigned to the variable will at least expect the same data type input and output.

A variable expecting a local function must be typed in the following fashion:

```plaintext
var functionVariable : ParamOne [-> ParamTwo] ...]--> ReturnValue;
```

Here, each of the function’s parameter types are provided and separated with an `->` operator, then, as functions can only return one value, that type is appended to the end and separated with yet another `->` operator.

Here’s an example:

```plaintext
var fun1 = function() : Void
{
    // code...
}
var fun2 = function( p1 : String ) : Void
{
    // code...
}
var fun3 = function( p1 : Int, p2 : Float, p3 : Int ) : String
{
    // code...
    return someString;
}
```

If you were to declare these variables before assigning the function references, you would do so like this:

```plaintext
var fun1 : Void -> Void;
var fun2 : String -> Void;
var fun3 : Int -> Float -> Int -> String;
```

In the first example, `fun1`, there are no parameters in the function definition, so `Void` is used. If, however, the function was to expect one or more parameters of type `Void`, they should be parenthesized, like this:

```plaintext
var fun1 : (Void) -> (Void) -> Void;
fun1 = function( p1 : Void, p2 : Void ) : Void
{
    // do code...
}
```
Part I: The Core Language

Passing Functions to Local Functions

One of the primary purposes of local variables is that they can be passed to other functions as parameters. Local functions can accept other local functions as their parameters. When doing so, the types of the parameterized function must be specified in the parameter type of the receiving function. This is done by parenthesizing each of the parameterized functions types so that they represent a single type:

```javascript
var fun1 : String -> String;
fun1 = function( p1 : String ) : String
{
    // do code...
    return someString;
}

var fun2 : String -> ( String -> String ) -> String;
fun2 = function( p1 : String, p2 : ( String -> String ) ) : String
{
    return p2( p1 );
}

var tmp = fun2( "someString", fun1 );
```

As you can see, this can appear a little similar to the parentheses applied to the Void types of functions. However, parenthesizing Void types of a local function only requires one type to be enclosed, while a full function definition will always require that two or more types are enclosed.

Multi-Function Recursion

Being able to type a variable to hold a function before it is assigned allows for some very interesting tricks. It is quite possible for two or more functions to make use of each other, and effectively create a recursive loop that spans across multiple functions. Tying functions together in this manner can prove very useful when parsing tree structures:

```javascript
var add2 : Int -> Dynamic -> Int;
var minus1 : Int -> Dynamic -> Int;
add2 = function( p1 : Int, p2 : Dynamic ) : Int
{
    p1 += 2;
    trace( p1 );
    if ( p1 < 5 ) p1 = p2( p1, add2 );
    return p1;
}

minus1 = function( p1 : Int, p2 : Dynamic ) : Int
{
    p1--;
    trace( p1 );
    p1 = p2( p1, minus1 );
    return p1;
}

add2( 0, minus1 );
// Outputs: 2, 1, 3, 2, 4, 3, 5
```

Here, one function adds 2 to a given variable while the other function deducts 1 from the variable until a certain limit is reached. As you can see from the example, the type of the functions for the parameters
was specified as Dynamic. This is because both functions take a function as their second parameter, which takes a function as their second parameter, which takes a function as their second parameter and so on, forming a recursive type specification. Leaving the typing to type inference results in an error, so assigning a Dynamic type is the only remaining suitable option. If you want to add some level of control over this type of function type definition, you could always specify the function type structure once for the variable, but specify the recursive function type as dynamic, ensuring that a function type is at least guaranteed to be passed on the first invocation. This will at least allow for proper usage of your function while delegating any further type checking to the object making the call.

**Local Function Variable Scope**

The variable scope for local functions includes static variables of the container class, variables that have been declared in the same level as the variable holding the function reference, and the variables that are declared inside the function:

```haxe
class LocalFunctionVars {
    public static var myStaticVar1 : String;
    public var myVar1 : String;
    public static function main()
    {
        var myLocalVar1 : String;
        var myFunction = function()
        {
            var innerFunction : String;
            innerFunction = "haXe";
            myLocalVar1 = "is";
            myStaticVar1 = "really";
            // will throw a compiler error
            myVar1 = "amazing";
        }
    }
}
```

This covers most of the variables in the class, but doesn’t account for the non-static variable myVar1. In order to access this variable, you need to reference it through the class keyword `this`. However, the `this` keyword in local variables is not allowed, as local functions are considered static, so a trick is necessary. If the variables local to the containing variable are accessible, even in a class instance, the value of `this` can be assigned to one such local variable and then referenced in the function:

```haxe
class LocalFunctionVars {
    public static var myStaticVar1 : String;
    public var myVar1 : String;
    public static function main()
    {
        var l = new LocalFunctionVars();
    }
    public function new()
    {
        var myLocalVar1 : String;
        var me = this;
```

(continued)
Part I: The Core Language

(continued)

```javascript
var myFunction = function()
{
    var innerFunction : String;
    innerFunction = "haXe";
    myLocalVar1 = "is";
    myStaticVar1 = "really";
    // will now compile
    me.myVar1 = "amazing";
}
}

Static variables and functions, and the this keyword are discussed in the next chapter. However, they have been detailed here for the purposes of completion.

Using Functions in Collection Objects

Earlier, when you looked at looping through an Array, you saw how, in order to directly modify the items stored in an Array while looping, the Array had to be indexed using an iterated integer value. However, what about List objects? The List collection cannot be indexed as the Array can, so how can the values contained in a List be modified in a loop? The answer is through the use of the List's map method.

The List map Method

The map method of the List collection applies a function to each and every item in its collection and returns a new List with the modified values. It is quite unlike any of the class methods you have looked at so far, as it takes a function as a parameter. This function must follow a set structure, because of strict typing, but can be created by using local functions. The type of the function, then, is:

```
var fun : X -> T;
```

The parameter, or X, is the type of the object contained in the original list, while the return value, or T, is the type of the object to be contained in the new List.

```javascript
class FunctionMapping
{
    public static function main()
    {
        var strLst : List<String> = new List();
        strLst.add("Bob");
        strLst.add("Burt");
        strLst.add("James");
        strLst.add("Rupert");
        var intLst : List<Int>;
        var fun : String -> Int = function( x : String ) : Int
        {
            return x.length;
        }
        intLst = strLst.map( fun );
    }
}
```
The List \texttt{strLst} was initialized and filled with a number of string values. You then constructed a local function, which received a value of type \texttt{String} and returned the length of the string as type \texttt{Int}. This function was passed to the Lists \texttt{map} method, which created a new List of type \texttt{Int} and, while iterating through the original List, applied the local function to each of the contained items and passed them to the new List. If the type being returned matched the type of the contained items in the original List, it would have made sense to assign the returned List to the original Lists variable, which would have provided the appearance that the values contained were directly affected.

**The List filter Method**

The List collections \texttt{filter} method works in a similar way to the \texttt{map} method, with the exception that, instead of applying a function to affect the values of the items contained in the List, the function passed to the method compares each item's value and returns a Boolean result. If the result of the function is true, then the item is returned in a new \texttt{List} object, otherwise it is excluded. This gives the effect, as you've probably already guessed, of filtering your values.

```cpp
class ListFiltering
{
  public static function main()
  {
    var strLst = new List<List>();
    strLst.add("Bob");
    strLst.add("Burt");
    strLst.add("James");
    strLst.add("Rupert");
    var newLst : List<String> = new List();
    var fun1 : String -> Bool = function(x : String) : Bool
    {
      return (x.length % 2 != 0);
    }
    var fun2 : String -> Bool = function(x : String) : Bool
    {
      return StringTools.startsWith(x, "B");
    }
    // Outputs: ["Bob", "James"]
    newLst = strLst.filter( fun1 );
    trace(newLst);
    // Outputs: ["Bob", "Burt"]
    newLst = strLst.filter( fun2 );
    trace(newLst);
  }
}
```
Part I: The Core Language

The Array sort Method

The Array sort method is similar to the List filter method in that it takes a function as a parameter and is used as a means to sort data. However, the function required by the Array sort method accepts two parameters, both of which should have the same type as the items in the Array, as both parameters will be two items from the Array. The return value of the function should be an Int of zero if the properties to compare of the two items are equal, greater than zero if the first parameter is larger than the second, and less than zero if the first parameter is less than the second. The items are then sorted inside the original Array by passing each item and its neighbor to the function and repositioning them appropriately. The items are rearranged inside the original Array, so the sort method returns Void.

```haXe
class ArraySorting
{
    public static function main()
    {
        var strArr = ["one","two","three","four","five","six"];
        var fun1 = function( a : String, b : String ) : Int
        {
            var x : String = a.charAt(0);
            var y : String = b.charAt(0);
            return if ( x == y ) 0 else if ( x < y ) -1 else 1;
        }
        var fun2 = function( a : String, b : String ) : Int
        {
            return if ( a == b ) 0 else if ( a < b ) -1 else 1;
        }
        // Outputs: ["four", "five", "one", "six", "two", "three"]
        strArr.sort( fun1 );
        trace(strArr);
        // Outputs: ["five", "four", "one", "six", "three", "two"]
        strArr.sort( fun2 );
        trace(strArr);
    }
}
```

As you can imagine, you can perform some very elaborate sorting methods on pretty much anything. Many languages have a single sort method that try to sort on some representation of the contained items, whereas haXe Array sort method provides ultimate control over how your data is organized.

The Lambda Class

The Lambda class contains several methods for modifying the content of Arrays and iterators in a fashion that is oriented toward a functional style of programming. To the greater extent, most of the methods are merely variations of methods accepting local functions as you saw earlier in this chapter, with the exception that they affect different collections than you’ve otherwise seen.

The Lambda class provides the list of methods as detailed Table 4-1.
Chapter 4: Controlling the Flow of Information

### Table 4.1

<table>
<thead>
<tr>
<th>Lambda Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array(A)( it : Iterable(A) ) : Array(A)</td>
<td>Creates (A) from Iterator(A)</td>
</tr>
<tr>
<td>count(A)( it : Iterable(A) ) : Int</td>
<td>Returns the number of elements in Iterator(A)</td>
</tr>
<tr>
<td>empty() ( it : Iterable(Dynamic) ) : Bool</td>
<td>Tells if an iterable does not contain any element</td>
</tr>
<tr>
<td>exists(A)( it : Iterable(A), f : A -&gt; Bool ) : Bool</td>
<td>Tells if at least one element of the iterable is found by using the specific function</td>
</tr>
<tr>
<td>filter(A)( it : Iterable(A), f : A -&gt; Bool ) : List(A)</td>
<td>Returns the list of elements matching the function (f)</td>
</tr>
<tr>
<td>fold(A, B)( it : Iterable(A), f : A -&gt; B, first : B ) : B</td>
<td>Performs the functional “fold” on Iterator(A)</td>
</tr>
<tr>
<td>foreach(A)( it : Iterable(A), f : A -&gt; Bool ) : Bool</td>
<td>Tells if all elements of the iterable have the specified property defined by (f)</td>
</tr>
<tr>
<td>has(A)( it : Iterable(A), elt : A, ?cmp : A -&gt; A -&gt; Bool ) : Bool</td>
<td>Tells if the element is part of an iterable</td>
</tr>
<tr>
<td>iter(A)( it : Iterable(A), f : A -&gt; Void ) : Void</td>
<td>Applies function ((X -&gt; T)) to all elements of Iterator(A)</td>
</tr>
<tr>
<td>list(A)( it : Iterable(A) ) : List(A)</td>
<td>Creates a List from Iterator(A)</td>
</tr>
<tr>
<td>map(A, B)( it : Iterable(A), f : A -&gt; B ) : List(B)</td>
<td>Creates an Iterator(B) from Iterator (A) with function ((X -&gt; T)) applied</td>
</tr>
<tr>
<td>mapi(a,b)( it : Iterable(A), f : Int -&gt; A -&gt; B ) : List(B)</td>
<td>Similar to map, but also passes an index for each item iterated</td>
</tr>
</tbody>
</table>

### map() And mapi()

The map and mapi methods of the Lambda class are very much like the map method of the List collection, with the exception that Lambda.map accepts an iterator as its first parameter and returns a new iterator, and Lambda.mapi performs the same feat, while passing an index to the processing function. Here is an example of these two functions in use:

```java
class Mapping
{
    public static function main()
    {
        var myArr : Array<Int> = [1, 2, 3, 4];
        var newIter : Iterator<Int>;
        var funIter = function( x : Int ) : Int
```
Part I: The Core Language

(continued)

```javascript
  { return x * 2; }
  newIter = Lambda.map( myArr, funIter );
  // Outputs: 2, 4, 6, 8
  for ( i in newIter )
    trace( i );
}
```

**array() And list()**

Creating an iterator from an Array or List is easy, thanks to the `iterator()` method they both expose. However, what about creating a List or an Array from an iterator? Luckily, this is exactly what `Lambda.array()` and `Lambda.list()` does. One of the great benefits to these methods is that it is quite quick and simple to convert an Array to a List and vice versa by using the methods as a proxy:

```javascript
  var myArr : Array<Int> = [ 1, 2, 3, 4 ];
  // Creates the List from the Array
  var myLst : List<Int> = Lambda.list( myArr );
  // Recreates the Array from the List
  myArr = Lambda.array( myLst );
```

**fold()**

The `fold` function is very useful, as it will allow you to produce a value from an iterator by applying a function that accepts each element in turn and returns a value based on a calculation with a given starting value. This value is then carried to the next calculation involving the next value in the iterator. For example, you could pass the `fold` function an iterator containing a list of integers, a starting integer value, and a function that takes two integers and adds them together. The result would then be the sum of all the items in the iterator plus the starting integer value:

```javascript
class FoldList
{
  public static function main()
  {
    var arr : Array<Int> = [1,2,3,4,5];
    var fun : Int -> Int -> Int = function( a : Int, b : Int ) : Int
    {
      return a + b;
    }
    var initialValue : Int = 22;
    // Outputs: 37
    trace( Lambda.fold( arr, fun, initialValue ) );
  }
}
```
Chapter 4: Controlling the Flow of Information

Summary

You now know much about program structure, data control, and the flexibility of functions in the haXe language. In this chapter, you mainly learned about:

- Using conditional statements to add logic to your applications
- Looping through elements in a collection
- Using iterators
- Applying functions to groups of elements in a collection.
- Applying functional programming methodologies to your haXe applications
- Using local variables and type declarations

In the next chapter, you expand on your knowledge of haXe program structures by being introduced to object-oriented programming in haXe. You also learn what the fundamentals of inheritance and polymorphism are, how to create your own iterable objects, and how to create objects that extend the Dynamic type.
Object-oriented programming (OOP) is a programming paradigm that enforces coding functionalities using objects as a metaphor. Developers seem to find in object representations a natural way to cope with abstract concepts, especially when the complexity of the system grows larger.

The OOP has always been seen in opposition to procedural programming that approaches coding as a sequence of steps to carry out. The latter is very practical in some specific context and until recently was the preferred choice for scripting languages such as VB and JavaScript, but the former has proven over the years to be an effective way to write codes easier to maintain and reuse.

haXe is a recent language and has embraced the OOP fully, but it still maintains a great degree of flexibility by adopting concepts of procedural programming.

In this chapter, you will learn:

- What classes and objects are and how can you use them in haXe
- What the differences are between instance fields and static fields
- How to declare and use variables and functions in classes
- How to extend classes and implement interfaces
- How to work with anonymous types, typedef and enum
- How to leverage on advanced features like extensions and type parameters
Part I: The Core Language

Classes and Objects

So, what are classes and what are they used for? Classes define a set of characteristics and implementations, while objects are their realizations. Figuratively speaking, a class is to an object as a bridge is to the Golden Gate. A construction must satisfy some requirements to fit the bridge definition: It must span a divide and have at least two access points; what differentiate the bridges are characteristics like the materials and their locations. The same applies in code where an object must satisfy the requirements of the class to be its instance and can have distinguishing characteristics.

The base syntax to declare a class is depicted in Figure 5-1.

```
class ClassName extends SuperClass, implements Interface
{
    //...
}
```

Figure 5-1

The class keyword denotes the declaration of a class. More than one class can be declared in a single file. Classes that are not prefixed with access modifier private are considered public and are accessible anywhere as long as they are imported correctly, as it is explained in the next chapter. Private classes are only available in their declaring file. The following class A is public while class B is private.

```
class A
{
    //...
}
private class B
{
    //...
}
```

A class name is a sequence of one or more alphanumeric characters, including the underscore character, starting with an uppercase letter; this convention is enforced by the compiler that will throw a compilation error otherwise. The class name is optionally followed by the keyword extends and by the class name of a parent class. Finally the class declaration can include one or more implements declarations. For each implements a single class name or interface name must be specified.

Features in classes are exposed as fields (also referred as members) of two kinds:

- **Variables**: placeholders for value containers, references for objects, or function definitions. Variables can be enhanced with access and value modifiers.
- **Functions**: actions that the object or the class can perform.
Chapter 5: Delving Into Object-Oriented Programming

Variables and functions can be declared as static fields or as an instance. A static field belongs to the class itself and is always accessible; an instance field is specific to an object and depends on its existence.

As an example, the implementation of a blog entry class in a hypothetical CMS (content management system) application is provided.

```haxe
class Main
{
    static function main()
    {
        var entry : BlogEntry = new BlogEntry();
        entry.title = "My First Blog Entry";
        entry.content = "Lorem ipsum ...";
        trace(entry.title);
    }
}

class BlogEntry
{
    public var title : String;
    public var content : String;
    public function new () { }
}
```

Two classes are defined; `Main` is just an entry point to test the execution of the code, while `BlogEntry` is part of the CMS business logic. `BlogEntry` just defines a data container for a bunch of information: the declaration of two variables, `title` and `content`, and a method `new` with an empty implementation.

Fields access, as shown in the `main` function, is performed using the classic dot-syntax as in many other languages like C/C++/C# and Java.

To compile the previous example, save the code in a single file named `Main.hx`, open the command line, go to the directory with the saved file and type the following command:

```
> haxe -main Main.hx -neko main.n
```

A file `main.n` will be generated. It can be executed in the command line where the results will be displayed, using the command:

```
> neko main.n
```

In this case, Neko is the execution target but any supported platform can be used in its place. All the examples in this chapter are compatible with any haXe platform unless otherwise indicated.
Part I: The Core Language

Static methods, like `main` in the previous example, will be covered later, but remember that when executing a piece of code written in haXe, there is always the need of a class implementing a static function named `main` that will be automatically executed on startup. The class containing such a method must be specified in the compiler command line using the `-main` switch. Compiling without the `-main` switch will build a library output; something that can be reused by other applications but that has no use by itself.

Instance Fields

Instance fields are characteristics of the concretization of a class in an object. Inside the class methods, the current object can be referenced using the keyword `this`.

Functions

Instance functions must be defined inside classes and follow the syntax described in Figure 5-2.

```
public function fName(al : Type, ?a2 : Type) : Type {
    //...
}
```

The first keyword is an optional access modifier that declares which is the accessibility of the field. If `public` is provided the function can be invoked from inside and/or outside the class declaration; if `private` or when omitted the field will be only accessible from inside.

```
class Main {
    static function main() {
        var sample : RestrictAccessSample = new RestrictAccessSample();
        trace(sample.publicMethod()); // works and traces 'private method'
        trace(sample.privateMethod()); // throws a compilation error
    }
}
```

class RestrictAccessSample {
    public function new () {
```
Chapter 5: Delving Into Object-Oriented Programming

```javascript
public function publicMethod() : String
{
    return privateMethod();
}

private function privateMethod() : String
{
    return "private method";
}
```

In the example `privateMethod` can only be invoked in the context of `SampleClass` while `publicMethod` is always accessible also inside other class declarations as long as it is associated to an instance of `SampleClass`.

Having private fields is necessary to transmit messages between functions without exposing them in the Application Programming Interface (API). This concept is known in OOP as *encapsulation*; the programmer that uses a certain class should not be aware of its internals.

**Null Arguments**

Assuming that a function argument should always provide a ready-to-use value is a common mistake. The following example shows why.

```javascript
class Main
{
    static function main()
    {
        var sample = new NullParameterSample();
        trace(sample.cube(7));
        trace(sample.cube(null));
    }
}

class NullParameterSample
{
    public function new () { }
    public function cube(n : Float) {
        return n*n*n;
    }
}
```

The method `cube` accepts a mandatory float parameter. If tested with value 7, the result is 343 as expected but if tested with `null`, which is a perfectly valid value on every platform but Flash 9, the result varies.

The best way to avoid inconsistencies is to verify the function parameters before using them and throw an error (error handling is covered in Chapter 7) or fallback to a common result.
Part I: The Core Language

The preceding example can be corrected this way to obtain a consistent result on every platform.

```haXe
public function cube(n : Float)
{
    return if(n == null)
        null;
    else
        n*n*n;
}
```

Optional Arguments

Every function argument admits an optional question mark `?` prefix. If present, it indicates that the argument is optional and it can be omitted when the function is invoked. Although optional arguments can be placed anywhere in a function declaration, it is usually a good practice to stack them at the end.

```haXe
class Main
{
    static function main()
    {
        var sample = new OptionalParameterSample();
        trace(sample.quote("quotation", " - "));
        trace(sample.quote("quotation"));
    }
}
class OptionalParameterSample
{
    public function new () { }
    public function quote(text : String, ?quotesSymbol : String) : String
    {
        if(quotesSymbol == null)
            quotesSymbol = """;
        return quotesSymbol + text + quotesSymbol;
    }
}
```

The second argument of the function `quote` is optional and when the function is invoked without the second argument, its value will be `null` by default. In practice, writing:

```haXe```
sample.quote("quotation");
```haXe```

is exactly the same thing as writing:

```haXe```
sample.quote("quotation", null);
```haXe```

For this reason, in the body of the `quote` function, the value of `quotesSymbol` is tested and, if it is `null`, an alternative default value is assigned. Remember that in haXe every variable type, even numeric types, can assume the value `null`; for this reason it's often prudent to test that a parameter is actually not a null value or unexpected errors may occur. When you want to enforce that a certain parameter can assume a null value, it is a better choice to make it optional. Flash 9 has a slightly different behavior regarding null values. Refer to Chapter 12 for further instructions.
Chapter 5: Delving Into Object-Oriented Programming

Optional parameters are independent: When more than one of them is declared in a row, it is possible to set a value for the last parameter without providing a value for the previous ones, although it can be a confusing practice. As an example consider this function:

```java
function foo(?a : String, ?b : Int, ?c : Person)
{
    // body goes here
}
```

All of the following statements are correct and correctly assigned to the right argument; arguments that are not explicitly stated are set to `null`.

```java
foo(1);
foo("John");
foo("John", 1, person); // where person is an instance of Person
foo(person);
```

But the following does not compile:

```java
foo(person, "John"); // wrong sequence
```

**Variables**

The formal definition of a class instance variable is described in Figure 5-3.

```java
private var name : Type;
```

Figure 5-3

The access modifier considerations made so far for instance functions can be applied here as well.

On the contrary to what happens for local variables and for static variables, it is not possible to set the value of an instance variable at the time of the declaration; the value can only be set inside a function call.

*Instance variables can be referenced inside functions simply by using their variable name or their variable name prefixed with the this term. The this identifier is mandatory inside a function that makes use of conflicting names.*

**Identifiers Sequence**

When naming variables and functions, it is important to be aware of how their identifiers are resolved. For variables or functions assigned to variables, the sequence in order of priority from highest to lowest, is:

- Local variables
- Instance variables
- Static variables
- `enum` constructors declared in the scope
Part I: The Core Language

Sequence for functions is the following:

- Functions stored in local variables
- Instance functions
- Static functions

If you have more than one local variable or function with the same name, the last declared before use will be adopted. Re-declaring a local variable or function is not really a good practice.

Instance class variables and functions can be from the current class or from a parent class as explained in the inheritance section of this chapter. Note that functions can be overridden but variables cannot.

The scope for `enum`s is the current file or every package imported; this argument is covered in Chapter 6.

Finally, you can have a static field and an instance field that share their name but you can’t have a class variable that shares its name with a class function.

**Constructor**

Every class can be instantiated only if it has a constructor that is a special function that is invoked every time a new instance is created. The syntax for the constructor is almost identical to that of the instance functions with the only exception for its name that must be mandatory `new`. The function return type is always `Void` and can be explicitly stated or omitted.

Because instance variables can’t be initialized in the same place of their declaration, the constructor should be used for this purpose.

To improve the `BlogEntry` class, a variable containing the creation date is added and its value is automatically generated at the instantiation time.

```javascript
class BlogEntry
{
    public var title : String;
    public var content : String;
    public var createdOn : Date;
    public function new ()
    {
        createdOn = Date.now();
    }
}
```

The constructor, as any other function, can have arguments that in the Blog examples can be used to set some constraints on the `BlogEntry` declaration and to reduce the line of codes required to create a usable instance of the class.

```javascript
class BlogEntry
{
    public var title : String;
    public var content : String;
    public var createdOn : Date;
    ...
}
```
Chapter 5: Delving Into Object-Oriented Programming

```java
public function new (title : String, content : String)
{
    this.title = title;
    this.content = content;
    createdOn = Date.now();
}
}

The instance is now created this way:

class Main
{
    static function main()
    {
        var entry : BlogEntry = new BlogEntry("My First Blog Entry", "...");
        trace(entry.title);
    }
}

Usually when a blog entry is created, it is just a draft and not really ready for publication. It is better to have a publish() method that expressly states when a post is ready to go online and a method unpublish(), to revert the post to the draft state just in case.

class BlogEntry
{
    public var title(default,default) : String;
    public var content(default,default) : String;
    public var createdOn : Date;
    public var onlineInfo(getOnlineInfo,null) : String;
    private var publishedOn : Date;

    public function new (title : String, content : String)
    {
        this.title = title;
        this.content = content;
        createdOn = Date.now();
        publishedOn = null;
    }

    public function publish() : Void
    {
        publishedOn = Date.now();
    }

    public function unpublish() : Void
    {
        publishedOn = null;
    }

    public function isOnline() : Bool
    {
        return publishedOn != null
                && publishedOn.getTime() <= Date.now().getTime();
    }
}

(continued)
Private function `getOnlineInfo()` : String

```javascript
private function getOnlineInfo() : String
{
    if(publishedOn == null)
        return "Not yet on-line";
    else
    {
        // getTime() returns the time in milliseconds since 1970-01-01
        // we have to divide the time span by 24 hours=24*60*60*1000=86400000
        var daysOnline = (Date.now().getTime()-publishedOn.getTime())/86400000;
        return if(daysOnline  <  1)
            "Published Today";
        else if(daysOnline  <  2)
            "Published Yesterday";
        else if(daysOnline  <  7)
            "Published " + Math.floor(daysOnline) + " days ago";
        else
            "Published on " + DateTools.format(publishedOn, "%Y-%m-%d");
    }
}
```

When the private variable `publishedOn` is set to `null`, the entry is not yet ready to be published and it is still a draft, as soon as the `publish()` method is invoked its value is set to the current date and, thus, ready to be displayed. The method `isOnline()` is used to check whether the entry has been published or not, while `getOnlineInfo()` returns information, in a human readable form, on the elapsed time since the date of publication.

**Variable Value Modifiers**

Class variables supports value modifiers, or properties, that permits to alter their standard behavior as follows:

- The access to the variable value can be limited to read-only or write-only.
- A getter and/or setter method can be declared to access the variable value.

The syntax for a modified variable is the one shown in Figure 5-4.

```
private var `name`<getModifier, setModifier> : `Type`;
```

The access modifiers have the same behavior as already explained for functions. The `Type` is the kind of value that the property can manage.
Chapter 5: Delving Into Object-Oriented Programming

The `getModifier` and `setModifier` can assume one of the values described in Table 5-1.

### Table 5-1

<table>
<thead>
<tr>
<th>Variable Value Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>null</code></td>
<td>The access is forbidden. Used in the <code>getKeyword</code> place, it means that the variable is accessible just for writing; using it in the <code>setKeyword</code> renders the variable read-only. Even if no errors are reported, having <code>null</code> in both positions makes no sense.</td>
</tr>
<tr>
<td><code>default</code></td>
<td>The variable is treated as a conventional variable. Declaring it with <code>default</code> in both <code>getKeyword</code> and <code>setKeyword</code> positions, is exactly the same as having an instance variable, but can enforce the semantic that the access modifier may need to be changed in the future.</td>
</tr>
<tr>
<td><code>method name</code></td>
<td>The specified method, usually marked as private, must belong to the same class of the property and is used to access the property value. If set in place of the <code>getKeyword</code> it will be a getter; otherwise it will be a setter.</td>
</tr>
<tr>
<td><code>dynamic</code></td>
<td>The access is provided by using a runtime-generated method.</td>
</tr>
</tbody>
</table>

The variable associated with a read-only/write-only property continues to be fully accessible inside the class that owns it. The access limitation is valid just from outside and makes sense for public variables only.

A getter method must conform to the signature `Void -> T`, which literally means that the method must accept no parameters and must return a value of type `T` where `T` is the type of the variable. The signature for the setter method is `T -> T`, which means that the function must accept and must return one value of the same type as the property itself.

The setter method, although it’s used for changing something inside the object instance, must always return a value (usually the same that would be returned accessing sequentially the getter method). This is needed to permit syntax like the following:

```javascript
var v = c.step++;  // Wrong!
```

Where `step` is a property defined this way:

```java
public var step(getStep,setStep) : Int;
private function getStep() : Int  
{  
  return counter;
}
private function setStep(value : Int) : Int  
{  
  counter += 1;
  return counter;
}
```

If `setStep` doesn’t return a value, `v` will be set to `Void` on the contrary of what one would logically expect.
The modifiers permit to limit the access to a variable from outside the class and this can be handy in the blog entry example to force the `createdOn` variable to be read-only so that its value is set at its creation and can’t be forged later.

```javascript
public var createdOn(default,null) : Date;
```

Trying to access the `createdOn` property will work fine if the value is read but reports an error if a value is assigned.

```javascript
class Main
{
    static function main()
    {
        var entry : BlogEntry = new BlogEntry("My First Entry", "...");
        trace(entry.createdOn); // works
        entry.createdOn = Date.now(); // fails
    }
}
```

To demonstrate another use of the read-only access, change the example to introduce a new variable `excerpt`, which will contain a small description of the blog entry using the first few words of the `content` property. The value is calculated on each call so that the class doesn’t require any support variable.

```javascript
public var excerpt(getExcerpt, null) : String;
private function getExcerpt() : String
{
    return content.substr(0, 10) + " ...";
}
```

The new code can be used like this:

```javascript
class Main
{
    static function main()
    {
        var entry : BlogEntry = new BlogEntry(
            "My First Entry",
            "Using the BlogEntry class is easy.");
        trace(entry.excerpt);
    }
}
```

The `excerpt` variable will now output the first ten characters of the `content` variable followed by a space and an ellipsis. In some cases it is better to provide an alternative text for the excerpt instead of cropping blindly the content of the blog entry. To obtain this, a setter method and a new private variable `definedExcerpt` are added to the class. The getter method is also modified to use the newly defined variable if it is different from `null` or recurring to the cropping functionality otherwise.

```javascript
private var definedExcerpt : String;
public var excerpt(getExcerpt, setExcerpt) : String;
private function setExcerpt(value : String) : String
{
    return value;
}
```
The excerpt property can now be set to a new value.

```javascript
entry.excerpt = "Blog Entry is easy";
```

In traditional OOP, getter and setter methods are the preferred way to access variables, but in dynamic languages such as JavaScript there is a general tendency to ignore this practice in favor of shorter, easier-to-read code and faster execution; with properties definition you can always switch with ease from instance variables to accessor methods.

The dynamic keyword in the context of variables is an advanced feature. To access the value of a property marked as dynamic, it is necessary to provide accessor methods at run time. The application itself must provide an implementation of the get/set methods dynamically. This can be done by modifying an object instance or the prototype of the class at run time using the reflection capabilities of haXe.

The property is defined in this way:

```javascript
var runtime(dynamic,dynamic) : String;
```

One or both the accessor methods must be provided dynamically and they must be named `get_runtime` and `set_runtime` (suffix `get/set_` followed by the property name), as shown in the following example:

```javascript
Reflect.setField(objectInstance, "get_runtime", function() {
  // do something here and return a string
  return "";
});
Reflect.setField(objectInstance, "set_runtime", function(value) {
  // do something here and return a string
  return value;
});
```

In practice the compiler does not check for the existence of the `get_X`, `set_X` methods and, thus they can be provided at run time by the developer.
Part I: The Core Language

The `objectInstance` is a reference to an object, which implements the class that defines the runtime property.

**Static Fields**

Static fields are characteristics that pertain to the class itself and not to its instances. A value set in a static variable will be the same whichever the context it is used in is and as long as the execution doesn't come to an end. They are conventionally used to store global values or to perform actions that rely only on the passed parameters. In a certain measure they replace the needs for global functions as encountered in procedural programming, with the plus of having a very clear context determined by the container class.

When a static field is referenced from inside the declaring class, it is possible to simply use its name unless it conflicts with other identifiers; in that case, and always when the field is used outside its declaring class, it is needed to prefix it with the class name, as in the following example.

```pascal
class Main
{
    static function main()
    {
        Sample.test(); // class name prefix is mandatory
    }
}
class Sample {
    public static function say() : String
    {
        return "Hello!";
    }
    public static function test() : Void
    {
        trace(say()); // same as trace(Sample.say());
    }
}
```

**Static Functions and Variables**

The syntax for static functions is equal to the one of instance function with the insertion of the keyword `static` between the access modifier and the `function` keyword.

The syntax to declare a static variable is the one depicted in Figure 5-5.

```
private static var name : Type = assignment;
```

Figure 5-5
Values for static variables can be assigned in the variable declaration; when this is the case it is possible
to omit the variable type because the compiler is capable of inferring the correct type from the assigned
value. Contrary to many other languages, the values for static variables are computed during the
execution bootstrap and not when they are first invoked.

Understanding Inheritance

As already explained, one of the biggest advantages in adopting OOP is about code reuse, a concept
nicely summarized in the DRY (Don’t Repeat Yourself!) acronym. A way to achieve code reuse is quite
simple in theory: When there are two or more classes that share common pieces of code, group them in
functions and move those functions to a base super-class.

Inheritance from another point of view can be seen as a hierarchy tree with a generic system at its root,
the concept of vehicle for example, and a more specialized system, like car or boat, on each branch; the
hierarchy can have an unlimited number of levels with several degrees of specialization. Every new level
in the hierarchy may add new functions and new variables: the class Vehicle should have the method
accelerate() which is inherited by all its descendants and the child class Car should have a variable
numberOfWheels that is not present in its sibling Boat.

After all, extending a class means creating a more specialized and feature-rich version of an existing class.

Every class can possibly extend no more than one parent. This is a quite common paradigm in
OOP known as single inheritance; haXe also provides alternative ways to extend as explained later in
this chapter.

An extended class inherits all the private and public instance fields but not the static ones. For people
coming from other OO languages like C# the private access modifier is equivalent to the protected
modifier.

Going ahead with the CMS example, a new article class is implemented. It has really a lot in common
with the BlogEntry class. They both have a title, a content, an excerpt and of course a way to be
published. So what? Must the code be repeated twice? Of course not. A common base is what is needed.
This is when inheritance comes into play.

Two new classes are then created: Article and BaseEntry. All the code written so far for the
BlogEntry class is moved to the BaseEntry. Article and BlogEntry are set to extend BaseEntry and
will inherit all its fields with no repetitions at all.

```hs
class BaseEntry {
  public var title(default,default) : String;
  public var content(default,default) : String;
  public var createdOn(default,null) : Date;
  public var onlineInfo(getOnlineInfo,null) : String;
  public var excerpt(getExcerpt, setExcerpt) : String;
  private var definedExcerpt : String;
  private var publishedOn : Date;
  public function new {title : String, content : String}
```
Part I: The Core Language

(continued)

```javascript
{ 
    this.title = title;
    this.content = content;
    createdOn = Date.now();
    publishedOn = null;
}
private function getExcerpt() : String
{ 
    if(definedExcerpt != null)
        return definedExcerpt;
    else
        return content.substr(0, 10) + " ...";
}
private function setExcerpt(value : String) : String
{ 
    definedExcerpt = value;
    return definedExcerpt;
}
private function getOnlineInfo() : String
{ 
    if(publishedOn == null)
        return "Not yet on-line"
    else
    { 
        // getTime() returns the time in milliseconds since 1970-01-01
        // we have to divide the time span by 24 hours=24*60*60*1000=86400000
        var daysOnline = (Date.now().getTime()-publishedOn.getTime())/86400000;
        return if(daysOnline  <  1)
            "Published Today";
        else if(daysOnline  <  2)
            "Published Yesterday";
        else if(daysOnline  <  7)
            "Published " + Math.floor(daysOnline) + " days ago";
        else
            "Published on " + DateTools.format(publishedOn, "%Y-%m-%d");
    }
}
public function isOnline() : Bool
{ 
    return publishedOn != null 
    && publishedOn.getTime() <= Date.now().getTime();
}
public function publish() : Void
{ 
    publishedOn = Date.now();
}
public function unpublish() : Void
{ 
    publishedOn = null;
}
}
class BlogEntry extends BaseEntry { }
class Article extends BaseEntry { }
```
Chapter 5: Delving Into Object-Oriented Programming

**Super**

When a class is extended, a new `super` identifier is accessible. It is very similar to the `this` identifier but refers to the parent class and not to the current one. As for the `this` identifier it can be omitted when there are no ambiguities. Consider this example:

```haxe
class Main
{
    static function main()
    {
        var test = new B();
        trace(test.getText());
    }
}
class A
{
    public function new() { }
    public function getAbc() : String
    {
        return "ABC";
    }
}
class B extends A
{
    public function getText() : String
    {
        return getAbc();
    }
}
```

The function call `getAbc()` is equivalent to `this.getAbc()` and to `super.getAbc()`.

**Functions Override**

When extending a class it can be useful to redefine the behavior of certain functions. To do so a new function with the same signature as the original (same access modifier, same name, same arguments, and same return type) is declared in the subclass.

To highlight that an already existing method has been redefined, the `override` modifier can be added in the function declaration just before the `function` keyword.

*The haXe compiler accepts an -override switch; when this switch is on, the compiler will throw an error whenever an overridden function without the override keyword is encountered. Leaving this switch always on is considered a good practice and helps to spot conflicting names.*

The `BlogEntry` and `Article` classes are now identical and share the same exact methods. What may distinguish the two is that `Article` represents a content, which is not a daily journal but more than an organic and detailed text — something the author has possibly dedicated much energy to and can take profit from a carefully planned publication. For this purpose the new method `publishOn` is added and will accept an arbitrary date of publication.
Part I: The Core Language

```java
class Article extends BaseEntry {
    public function publishOn(date : Date) : Void {
        publishedOn = date;
    }
    private override function getOnlineInfo() : String {
        return if(publishedOn != null &&
            publishedOn.getTime() > Date.now().getTime())
            "Not yet on-line, scheduled for " +
            DateTools.format(publishedOn, "%Y-%m-%d");
        else
            super.getOnlineInfo();
    }
}
```

*Inside a redefined function, if you want to call the overridden definition it is necessary to use the super identifier.*

The method `getOnlineInfo()` has been redefined to accommodate the extra functionality introduced. In the `true` part of the `if` statement, a new case for a future planned publication date has been introduced while the `false` condition recalls the original function as defined in the `BaseEntry` class.

Having multiple classes that share the same function definitions with different implementations is known in OOP as *polymorphism*.

**Constructor Override**

The use of the `super` identifier in a constructor is slightly different from what you have already seen for invoking functions in a parent class. First of all to call the constructor of a super class the `super` identifier is used directly as a function; then the constructor signature can be completely different from parent to child. That means that a base class that has a constructor with no arguments can be redefined in a subclass using one or more arguments or that a private constructor can be changed in a public constructor.

The use of the modifier `override` is not allowed in the declaration of the constructor.

The `Article` class from the previous example can be modified to take advantage of a redefined constructor:

```java
public function new (title : String, content : String, ?date : Date) {
    super(title, content);
    if(date != null)
        publishOn(date);
}
```
Chapter 5: Delving Into Object-Oriented Programming

**toString()**

Every value in HaXe, whatever its type, can be automatically converted into a string when required. The result of the conversion depends on the type of the value and on the platform in which the conversion is executed. For primitive types such as *floats* or *integers*, the result obtained is always consistent, but in the case of class instances the result may vary greatly. If an instance of the BlogEntry class is passed as an argument to the trace function, in Neko the output will be something like this:

```
Main.hx:7: { createdOn => 2007-05-02 06:08:02, publishedOn => 2007-04-20 01:00:00, title => My First Article, content => My Content }
```

Whereas Flash 9 will output:

```
Main.hx:7: [object BlogEntry]
```

To change this behavior, it is possible to simply implement the `toString()` method like this:

```
public function toString() : String
{
    return title;
}
```

And the result will be consistently the same for every platform.

**Abstract Classes and Abstract Methods**

In many OO languages exists the formal definition for abstract classes. An abstract class cannot be instantiated by itself and must necessarily be sub-classed to be of some use.

This could be the case in your `BaseEntry` class in the blog example. The class is there just to provide a common base for `Article` and `BlogEntry` and instances of `BaseEntry` should not really be created. To forbid this possibility and to obtain a sort of abstract class, the access modifier of the constructor of `BaseEntry` is changed to private.

```
private function new (title : String, content : String)
```

Now trying to instantiate the `BaseEntry` class throws an error at compile time.

Sometimes it is useful to have the abstract concept applied to methods. In this case the base class just declares the function signature and delegates the actual implementation to its descendants. To enforce this type of constraint the base class defines a function that always throws an error; if this function is not overridden in its descendants, the code execution will be blocked by the thrown error.

```
class Animal
{
    public function speak() : String
    {
        throw "abstract method, provide an implementation in the sub-class";
    }
}
```
Part I: The Core Language

(continued)

    return null;

class Dog extends Animal
{
    public function new() { }
    public override function speak() : String
    {
        return "bark!";
    }
}

class Cat extends Animal
{
    public function new() { }
    public override function speak() : String
    {
        return "meow!";
    }
}

Error handling is discussed in detail in Chapter 7; for now, it is enough to know that throw just creates an error in the code flow. Also, in this example the entire Animal class is abstract and cannot be instantiated because it does not contain a constructor at all.

Static Fields, Instance Variables, and Inheritance

Static fields are not inherited and, thus, cannot be overridden because they are not part of the class prototype. Instance variables are inherited but cannot be overridden.

In the case of static functions, this does not mean that a descendant class can’t declare a function already existing in one of its ancestors. Simply, the two are completely unrelated in their signature; it does not exist in any super identifier for static functions.

Instance variables are always referenced using the this prefix (or nothing if there is no ambiguity), regardless if they were declared in the current class or in any of its ancestors.

Inheritance Roles

When considering the inheritance hierarchy, it’s easy to figure out that the same instance can play different roles at the same time. Considering an instance of the BlogEntry class, it is at the same time an object representing a blog post and a more generic entry entity. In practice it is possible to deal with the instance at a different level of specialization. In code this can be used to further avoid code repetition.

Continuing the CMS example, imagine that there is the need of a page in the back end that shows a description and the status of all the contents in the repository, no matter if they are articles or blog posts. Instead of having one list for each type of container, it is possible to have a unique list of type BaseEntry.
Chapter 5: Delving Into Object-Oriented Programming

```javascript
var list = new List<BaseEntry>();
list.add(new Article("Article #1", "...", Date.now()));
list.add(new BlogEntry("Blog Entry #1", "..."));
list.add(new Article("Article #2", "...", Date.fromString("2020-01-01")));
for(entry in list)
    trace(entry.title + " [" + entry.onlineInfo + "]");
```

It is possible to easily manipulate the list using the fields provided in the BaseEntry class declaration. If for example, it is needed to create a list with just the published entries, it is easy to apply a filter to the original list and to obtain a reduced copy:

```javascript
var filteredList = list.filter(function(entry)
{
    return entry.isOnline();
});
```

Note that in the passed function you do not have to mandatorily declare the type of the argument entry because the compiler can infer it from the usage context.

### Using Interfaces

An interface is a type that is completely abstract; it provides the skeleton (instance field definitions) but no implementation at all. Because it does not provide any routine, it cannot be instantiated. For this reason classes can only implement interfaces and not extend them. A class implementing an interface must mandatorily provide a definition for each field contained in the interface.

The interface declaration is very similar to the class declaration and uses the same naming constraints; see Figure 5-6 for its syntax.

```javascript
interface IPerson
{
    var firstName : String;
    var familyName : String;
    function getFullName(): String;
}
```

The interface body can contain function and variable definitions as it happens for the class definitions with a notable difference: Function cannot have a body as shown in the following example.

```javascript
interface IPerson
{
    var firstName : String;
    var familyName : String;
    function getFullName(): String;
}
```
Part I: The Core Language

The `I` prefix in the interface name is a naming convention in some other languages such as C#, but it is not required and you can follow your own style conventions or none at all.

An interface can implement one or more interfaces but cannot extend none. On the contrary of what happens in a class definition, when the access modifier is omitted the default value of `public` is assumed; the types for functions, parameters, variables, and properties must always be explicitly provided.

Static fields have no meaning in interface declaration because, as already explained, they are owned just by the declaring type and are not inherited.

Interfaces in practice are used when a system requires exchangeable classes, which share their definition but not their implementation. Such a system could be for example a graphical user interface (GUI) that requires that its widgets (buttons, text fields, and so on) implement a common `Control` interface.

Another typical use case for interfaces is in the definition of a layer of abstraction over a low-level system; in haXe this type of abstraction is found in the `neko.db.Connection` interface that defines a generic and interchangeable approach to connect to a database. All databases, independently of their origin, perform some common tasks: executing queries, connecting to a data source, and so on. Those common tasks are implemented in very different ways that depend on the vendor choices. To unify the behaviors of different libraries and, thus realize the layer of abstraction, it is necessary to write some glue codes. This code will contain functions that wrap the original API and map those functionalities in a class implementing the `neko.db.Connection`. In this way, it is possible for example, to use MySQL and SQLite exactly in the same way and more importantly in an exchangeable way. When it is needed to switch from a db to another, it is enough to change the instantiation line.

Advanced Classes and Objects Features

Sometimes accessing methods and variables directly is not a viable solution; you may need more flexibility in accessing private members or you may not know which field has to be accessed until a few moments before that happens. To resolve those extreme situations you may rely on the reflection API that is described in detail in Chapter 16.

In the blog entry example, it is not too difficult to spot that the implementation of the `createdOn` property is a little bit pointless in the context of a CMS application that uses a database to store and retrieve the site contents. It is true that creating records in a database table by reading the properties and the variables one by one is quite easy. But what happens when the object must be loaded from the database? How is it dealt the assignment of the `createdOn` property because it is read-only?

Reflect and Type classes can offer a solution to the problem, as shown in the following example:

```java
var entry : Article = Type.createEmptyInstance(Type.resolveClass("Article"));
Reflect.setField(entry, "createdOn", Date.fromString("2007-04-10"));
```

The class is instantiated using the `Type.createEmptyInstance()`, which builds an object without even invoking the constructor (in this case there would not be a great difference in instantiating the object.
using the new keyword, but there are cases where this can make really a great difference). The `Reflect
.setField()` method is then used to set the read-only property `createdOn`. The remaining variables
can be set normally.

**Implementing Classes**

*Implementation* is the typical relation that exists between a class and an interface; haXe is much more
flexible and also permits that classes may implement other classes. When implementing another class it
will be necessary to provide code for each of the instance fields associated with the implemented class,
both public and private, and no code will be automatically borrowed in the same way as it happens
when implementing interfaces.

*The Flash 9 platform doesn’t support the implementation of classes at all. On this platform it is possible
just to extend one class and to implement one or more interfaces.*

**Type Parameters**

Some classes or interfaces are specialized in operating over instances of other classes. Such operations
can possibly be performed on instances of classes that do not share any common parent class or
interface. Consider the following example:

```haxe
class BinaryNode {
    private var content : String;
    public var parent : BinaryNode;
    public var leftChild : BinaryNode;
    public var rightChild : BinaryNode;
    public function new (content : String)
    {
        this.content = content;
    }
    public function getContent() : String
    {
        return content;
    }
}
```

The `BinaryNode` class is a very simple container for hierarchically stored information. In this
example, the content of each node is of `String` type but it sounds quite obvious that exactly the same
node class can easily be adapted to store any kind of value. To do that and with the knowledge acquired
so far, the problem can be solved in one of the following two ways: declaring a new node class for each
type you want to store in it or using a content of `Dynamic` type. The first solution is tremendously
repetitive (remember DRY: Don’t Repeat Yourself!) and a real pain when the implementation just
changes a little bit. The second solution is weak because it is necessary to continually *cast* to a proper
type each time the `getContent()` is invoked, and it is always a good practice to avoid *unsafe casts* if not
absolutely necessary.

What is needed is a generic approach (*generics* is the term used in the same context in languages like C#
and Java); this is exactly what type parameters are used for. A type parameter is a generic placeholder for
a concrete type.
Part I: The Core Language

The example can now be re-coded as follows:

```java
class BinaryNode<T> {
    private var content : T;
    public var parent : BinaryNode<T>;
    public var leftChild : BinaryNode<T>;
    public var rightChild : BinaryNode<T>;
    public function new (content : T) {
        this.content = content;
    }
    public function getContent() : T {
        return content;
    }
}
```

On class instantiation a type must be provided in place of T in this way:

```java
var ns : BinaryNode<String> = new BinaryNode<String>("I'm a node");
```

Another example could be:

```java
var ni : BinaryNode<Int> = new BinaryNode<Int>(7);
```

Now `ns.getContent()` will return a value of `String` type while `ni.getContent()` will return a value of `Int` type.

You can shorten the type declaration, further taking profit from type inference and reduce the code to this:

```java
var ns = new BinaryNode("I'm a node");
```

The `String` type parameter can be omitted in the constructor because the compiler can infer the correct type from the textual argument. The variable does not need any type declaration because the complete type can be inferred by the constructor.

The type declared at the moment of the object creation will always remain the same until the object is garbage collected, and cannot be changed at run time.

Giving the name T to the class parameter is just a convention. More than one class parameter can be specified using a comma separated list of names between angular brackets.

Type parameters can also be specified at function level; in this case the generic parameter will have a purpose only in the context of the declaring function and is declared in the same way class type parameters are but located just after the function name.
Chapter 5: Delving Into Object-Oriented Programming

```haxe
public static function indexOf<T>(v : T, a : Array<T>)
{
    for(i in 0...a.length)
        if(v == a[i])
            return i;
    return -1;
}
```

This function takes as arguments a value of type \( T \) and an \( \text{Array} \) of the same type and returns the position of the value in the array or \(-1\) when not found. The function is used in the conventional way:

```haxe
var l = [1,2,4,8,16]; // l is of type \( \text{Array} \)<Int>
trace(indexOf(8, l));
```

That traces \( 3 \) or with another kind of array:

```haxe
var ls = [*A*,*B*,*C*,*D*]; // ls is of type \( \text{Array} \)<String>
trace(indexOf(*C*, ls));
```

It traces \( 2 \).

**Constraints on Type Parameters**

Being generic can be useful; being too generic can be confusing. For this reason, \texttt{haXe} provides a way to restrict the class parameters to some types only. Constraint parameters use the syntax illustrated in Figure 5-7.

```haxe
class Name<T : Type, T : \langle Type, Type\rangle>
{
    access modifier
    multiple constraints for the same type parameter are grouped in parenthesis
    class Name<T : Type, T : \langle Type, Type\rangle> (continued)
}
```

The use of rounded parentheses is mandatory to disambiguate when more than one class parameter is used. Multiple constraints are usually the combination of one class and one or more interfaces; the type \( T \) must satisfy all of the listed constraints. The type constraints are not limited to classes and can also be applied to type parameters for interfaces, \texttt{enums}, and \texttt{typedefs}.

When you use type constraint, it is possible to realize specialized object containers such as the one illustrated in the example.

```haxe
class Item {
    public function new() {}()
}
class Movie extends Item {}
class Butterfly extends Item {}
class Collection<T : Item>
(continued)
```
Part I: The Core Language

(continued)

```java
{ public function new() {} public function add(item : T) { // implementation goes here } }
```

In your application you can use the `Collection` type to create a container just for movies or butterflies.

```java
var movies = new Collection<Movie>(); // the constraint is on the class Movie
movies.add(new Movie()); // accepted value
movies.add(new Butterfly()); // compiler does not permit this
```

Constraints are also useful with standard types. A constraint on Float can limit the accepted values to numbers; at instantiation it is possible to state if the value must be integer or real.

```java
class Point<T : Float>
{
    public var x : T;
    public var y : T;
    public function new(x : T, y : T)
    {
        this.x = x;
        this.y = y;
    }
}
```

The class can then be used in the following way:

```java
var pInt = new Point<Int>(10, 20); // pInt.x = 0.1; // does not compile
var pFloat = new Point<Float>(0.1, 0.2);
```

**Anonymous Objects**

Anonymous objects are structures of data created without instantiating any particular class. They are recognized in the haXe type system as pertaining to the anonymous type.

Their declaration is very short because it requires just the identifier’s fields and their values:

```java
var color = { r: 255, g: 255, b: 255 };
```

The type of color object is:

```java
{ r:Int, g:Int, b:Int }
```

You can declare the expected type this way:

```java
var color : { r:Int, g:Int, b:Int } = { r: 255, g: 255, b: 255};
```
Chapter 5: Delving Into Object-Oriented Programming

You can also reuse compatible typedefs like so:

```java
var color : { r:Int, g:Int, b:Int, a:Int } = { r:255, g:255, b:255, a:100 }
var color2 : { r:Int, g:Int, b:Int } = color;
```

Note that the second type definition contains a field less than the assigned object. This is perfectly legal because the type definition just imposes the minimum requirements. To create an empty anonymous object, (an object with no defined fields) use the method `Reflect.empty()` and not `{}`. The second statement is in haXe the declaration for an empty block of code that returns `Void` and not an empty object.

**Implementing Dynamic**

The use of the `Dynamic` type is discussed in Chapter 3; here you explore how `Dynamic` can add yet more flexibility to standard classes.

First of all any newly defined class can implement `Dynamic`. When this is the case, the instance class fields are strongly typed if defined in the class. If not, they are of the `Dynamic` type. Consider the following example:

```java
class Main
{
    static function main()
    {
        var item = new Item();
        // regular access to class field
        item.name = "My Item";
        // now exists a description field with the specified value
        item.description = "Item description";
        // would fail because the type of price is defined in the class and
        // must be a numeric value not a string
        //item.price = "-";
        // trace to null because a dynamic object contains every possible field
        trace(item.quantity);
    }
}

class Item implements Dynamic
{
    public var name : String;
    public var price : Float;
    public function new() {}
}
```

**Dynamic Type Parameters**

`Dynamic` can also take one type parameter; in that case every field of the `Dynamic` object will have the type of the passed parameter. This is a fast and effective way to implement a hash table that makes use of the dot syntax.

```java
var table : Dynamic<String> = cast Reflect.empty();
table.a = "A"; // works just fine
table.b = 1; // throws a compilation error
```
Part I: The Core Language

Finally a class can implement Dynamic with a type parameter. In this case every field explicitly defined in the class will maintain its own type, while all the others will have the type of the specified class parameter.

When a class implements Dynamic, all the derived classes will inherit Dynamic, too. In case one or more classes implement Dynamic with a class parameter, the last declared in the chain will be the effectively used one.

**Typedef**

Typedef is a construct for defining types that are used for type checking of anonymous types. Typedef declarations must use the following syntax:

```plaintext
typedef Name = typedeclaration
```

Where Name must follow the same rules exposed for class names and typedeclaration is the signature of the type definition.

The most common use of typedef is to give a formal representation to anonymous objects:

```plaintext
typedef Color = { r: Int, g: Int, b: Int}
```

Used like this:

```plaintext
var white : Color = { r: 255, g: 255, b: 255 }; 
```

The typedef can also be used to define other types, such as functions or shortcuts to existing definitions:

```plaintext
typedef GenericFunction<T> = Void -> T
typedef IntArray = Array<Int>
typedef P = Person
typedef TS = ThreeState
enum ThreeState
{
    Checked;
    Unchecked;
    Indetermined;
}
class Person
{
    public var name : String;
    public function new() {} 
}
```

When declaring the type of an anonymous object, you have two syntax possibilities:

```plaintext
typedef A = {
    var x : Float;
    var y : Float;
}
```
Chapter 5: Delving Into Object-Oriented Programming

```haxe
typedef B = {
    x : Float,
    y : Float
}
```

In this case, the declarations are equivalents; but you can see the difference with a `typedef` that contains at least a function definition:

```haxe
typedef A = {
    function say(text : String) : String;
}
typedef B = {
    say = String => String
}
```

In this case, the two are almost identical with the only difference being that the function argument is named in the first declaration and anonymous in the second. Another difference in the two syntax forms is that in the first form you can add the private/public access modifier (by default it is always public).

The `typedef` can also have type parameters as for classes and interfaces.

```haxe
typedef Leaf<T> = {
    parent : Node<T>,
    data : T
}
```

The `typedef` syntax is an important feature of the haXe language and one that makes haXe unique. `typedefs` are in many ways a better and more flexible alternative to interfaces. The `typedef` verification is made only at compile time and it is structural; the compiler checks that the passed object has the required fields but makes no assumptions on the type definition of the object. That creates very interesting possibilities. In the following example the function `handle()` receives as an argument a value of type `HasName`. `HasName` is a `typedef` that defines that the value of its type must have one field name of type `String`. The nice thing is that not only anonymous objects can satisfy this structural requirement but also class instances.

```haxe
class Person
{
    public var name : String;
    public function new(n : String)
    {
        name = n;
    }
}
class Item
{
    public var name : String;
    public var price : Float;
    public function new(n : String, p : Float)
    {
        name = n;
        price = p;
    }
}
```

(continued)
Part I: The Core Language

typedef HasName = {
    name : String
}

class Main {
    public static function handle(o : HasName)
    {
        trace(o.name);
    }
    public static function main()
    {
        var person = new Person("John");
        var item = new Item("Laptop PC", 999.9);
        var user = { name : "Jane" };
        handle(person);
        handle(item);
        handle(user);
    }
}

You can even simplify the declaration removing the typedef definition and replacing the handle declaration with the following line:

public static function handle(o : { name : String})

types are not embedded in the generated output but only used for type checking at compile time.

They enforce a kind of programming based on conventions where developers are required to build structures with certain characteristics but are not bounded to formal definitions like interfaces or base classes. You can define a convention for a project and without providing any code to a remote colleague implement a compatible solution with him. This can be of great benefit in the web development environment where it is usual that many libraries are supposed to work together to outfit the final product.

The same applies not only to objects but to any type definition. It is interesting for example having a function that accepts another function as an argument; the argument can be an inline function, an object method, or a class static function. Here’s an example:

class StringDecoration
    {
        var prefix : String;
        public function new(prefix : String)
        {
            this.prefix = prefix;
        }
        public function decorate(s : String)
        {
            return prefix + s;
        }
    }
class Main
Chapter 5: Delving Into Object-Oriented Programming

```plaintext
{ public static function print(s : String, decorate : String -> String)

  var o = decorate(s);
  trace(o);
}

public static function quote(s : String)
{
  return '"' + s + '"';
}

public static function main()
{
  var decorator = new StringDecoration("-> ");
  var s = "John";
  print(s, quote); // traces "John"
  print(s, decorator.decorate); // traces -> John
  print(s, function(s){ return "-- " + s + " --"; }); // traces -- John --
}
}
```

The second argument of the `print()` function is a typedef for a function that accepts a `String` as an argument and returns a `String`. In the `main()` method the `print()` function is used passing the static function `quote()`, then the instance function `decorate()` of the class `StringDecoration`, and finally an inline function definition.

**Iterators and Iterables**

In the haXe standard library, two very commonly used typedefs are defined: `Iterator<T>` and `Iterable<T>`. Their definition is as follows:

```plaintext
typedef Iterator<T> = {
  function hasNext() : Bool;
  function next() : T;
}

typedef Iterable<T> = {
  function iterator() : Iterator<T>;
}
```

As you have already seen in Chapter 3, those structures are defined to deal with object collections and they are easily implemented in class definitions. In the example, a random iterator is defined. The iterator will return a sequence of \( n \) random letters.

```plaintext
class RandomSequence
{
  private var letters : Int;
  private var counter : Null<Int>;
  public function new(letters : Int)
  {
    this.letters = letters;
  }
}
```

(continued)
(continued)

```haxe
public function hasNext() : Bool
{
    if(counter == null)
    {
        // the iterator has to be initialized
        counter = 0;
    }
    if(counter  <  letters)
    {
        return true;
    } else {
        // before returning false, the counter variable must
        // be reset in case the instance has to be used again
        counter = null;
        return false;
    }
}

public function next() : String
{
    counter++;
    return Std.chr(Std.int(Math.random() * 26) + 65);
}
}
```

The code doesn’t declare explicitly that it is implementing the `Iterator` typedef. This implicitly happens because the class defines the correct methods. The class is used like this:

```haxe
class Main
{
    static function main()
    {
        var sequence = new RandomSequence(10);
        for(letter in sequence)
        {
            trace(letter);
        }
    }
}
```

The example will trace ten times a random letter.

In practice what haXe does behind the scene is transforming the `for(letter in sequence)` declaration in:

```haxe
while(sequence.hasNext())
{
    var letter = sequence.next();
    // ...
}
```
Chapter 5: Delving Into Object-Oriented Programming

This automatically happens every time an object conforming to the `Iterator<T>` typedef is used in a `for` statement. The same treatment is reserved for the objects that conform to the `Iterable<T>` typedef. This can be useful when the object can return a collection but with an intermediate level of indirection.

The `Array<T>` is `Iterable<T>` and can be used to demonstrate its use. The following two declarations are in fact identical; the compiler internally changes the second to behave like the first.

```plaintext
for(i in [1,2,4,8].iterator())
{
    trace(i);
}
for(i in [1,2,4,8])
{
    trace(i);
}
```

**Extensions**

Extensions are a way to extend classes or anonymous object typedefs on the fly. They are rarely used in conjunction with classes, because the classic inheritance is usually preferred.

The syntax is the same in both cases where `type` can be a class or a typedef:

```plaintext
variable = { > Type, /* new field definitions goes here */ }  
```

When using extensions with classes, you will always have to cast the instance to the extended type because classes do not define types; always pay attention when using casts this way because they are *unsafe casts*.

```plaintext
class Main
{
    static function main()
    {
        var p : Contact = cast new Person();
        p.name = "John Doe";
        p.telephone = "123456";
    }
}
class Person
{
    public var name : String;
    public function new() {}
}
typedef Contact = { >
    Person,
    telephone : String
}
```

*Class instances in Flash 9 cannot be dynamically augmented (adding fields at run time) unless they also implement Dynamic.*
Part I: The Core Language

Extensions applied to `typedef` have exactly the same structure:

```java
typedef Color = {
    r: Int,
    g: Int,
    b: Int
}
```

```java
typedef AlphaColor = {
    Color,
    a: Int
}
```

All the objects of type `AlphaColor` are of the type `Color`, too; in this aspect typedef extensions are very similar to the standard class inheritance where an instance of a derived class is also an instance of the parent class.

**Enum**

Almost every other language implements a form of enumeration as a way of defining a list of fixed symbols. In haXe this functionality is provided using the `enum` syntax. Usually in other languages enumerations are mapped to some numeric integer value. haXe is not limited to that: `enums` are a first class type definition and represent a type with a fixed set of constructors (zero or more) where constructors can also have arguments.

The correct syntax for `enum` is illustrated in Figure 5-8.

```
enum Name {
    firstConstructor;
    otherConstructor(a1 : Type, a2 : Type);
}
```

**Figure 5-8**

The `enum` name follows the same convention of class names.

In their simplest form, they are adept at restricting a selection to a certain set of values:

```java
class Main {
    static function main() {
        trace(ByteUnitTools.humanize(1100, Kilobyte)); // trace "1.07 megabyte(s)"
    }
}
```
Chapter 5: Delving Into Object-Oriented Programming

```java
enum ByteUnit {
    Byte;
    Kilobyte;
    Megabyte;
}

class ByteUnitTools {
    public static function getBytes(unit : ByteUnit) : Int {
        return switch(unit) {
            case Byte:     1;
            case Kilobyte: 1024;
            case Megabyte: 1024 * 1024;
        }
    }

    public static function humanize(value : Int, unit : ByteUnit) : String {
        var bytes = getBytes(unit) * value;
        return if(bytes  < = getBytes(Kilobyte))
            bytes + " byte(s)"
        else if(bytes  < = getBytes(Megabyte))
            round(bytes / getBytes(Kilobyte)) + " kilobyte(s)"
        else
            round(bytes / getBytes(Megabyte)) + " megabyte(s)";
    }

    private static function round(value : Float) {
        return Math.round(value * 100) / 100;
    }
}
```

Note that you can use a constructor identifier directly without any prefix; the type name prefix must be used only in case of ambiguity.

---

Switch and enum have a great affinity when working together. If you add a gigabyte to ByteUnit and try to compile again, a compilation error will be raised. This happens because the switch statement requires that all the possible cases of an enum have to be contemplated. To bypass the problem it is necessary to provide a new case in the switch statement or alternatively a fallback default case. This behavior helps detect errors and inconsistencies at an early stage.
Part I: The Core Language

**Constructor Arguments**

The `enum` constructors can have arguments just as functions have. To show that feature, a simple query builder that takes advantage of the strongly typed syntax of haXe is defined as follows:

```haxe
enum Condition
{
    TestIsNull(field : String);
    TestNumber(field : String, operator : NumericOp, value : Float);
    TestText (field : String, operator : TextOp, value : String);
    And       (tests : Array<Condition>);
    Or        (tests : Array<Condition>);
}
```

```haxe
enum NumericOp
{
    Equal;
    Different;
    MoreThan;
    LessThan;
}
```

```haxe
enum TextOp
{
    Same;
    Like;
}
```

The `enum` `Condition` contains the constructors for each possible test that a query can define (more definitions are possible, but the list is kept short for sake of clarity). Note how the `and` and `or` constructors are recursive, as they use `Condition` in their parameters. The following two enumerations represent the operators used when comparing a field with a value.

What is needed now is a utility class to manipulate your `enum`s.

```haxe
class QueryTools
{
    static public function toSql(q : Condition) : String
    {
        return switch(q)
        {
            case TestIsNull(field):
                field + " IS NULL";
            case TestNumber(field, operator, value):
                field + " " + getNumericOp(operator) + " " + value;
            case TestText(field, operator, value):
                field + " " + getTextOp(operator) + "'"
                + StringTools.replace(value, ",", ",\"\") + ";"
            case And(tests):
                join(tests, "AND")
            case Or(tests):
                join(tests, "OR")
        }
    }
}
```
Chapter 5: Delving Into Object-Oriented Programming

private static function join(tests : Array<Condition>, op : String) : String
{
    return "(" + Lambda.map(tests, toSql).join( " + op + " ) + ");"
}
private static function getNumericOp(op : NumericOp) : String
{
    return switch(op)
    {
        case Equal: "=";
        case Different: "!=";
        case MoreThan: " > ";
        case LessThan: " < ";
    }
}
private static function getTextOp(op : TextOp) : String
{
    return switch(op)
    {
        case Same: "=";
        case Like: "LIKE";
    }
}

The syntax to extract the parameters from a constructor in a switch statement is the following:

switch(name)
{
    case Constructor(p1, p2): /* do something here */;
}

The param has no type specified because the type is inherited directly from the constructor definition.

It is now possible to use the enum this way:

class Main
{
    static function main()
    {
        var t = And(["TestText('name', Same, 'John'),
                      TestText('lastname', Same, 'Doe'),
                      Or(["And(["TestIsNull('age'),
                         TestText('notes', Like, '%authorized%'),
                     ]),
                      TestNumber('age', MoreThan, 18)
                ]));
        trace(QueryTools.toSql(t));
    }
}
Part I: The Core Language

That will nicely trace:

```
(name = 'John' AND lastname = 'Doe' AND ((age IS NULL AND notes LIKE '%authorized%') OR age > 18))
```

*It is important to note that using switch is the only available way to extract the values associated to a constructor without having to recur on Reflection (see Chapter 16 for details).*

As already seen for other types, enums can also use type parameters. Here’s a short example to point it out:

```
enum ValueStatus<T>
{
    NonExistent;
    Unknown;
    Known(value : T);
}
```

Take a look at Chapter 16 in the “Reflection API” section to learn how to retrieve the index of an enum constructor; it is quite obvious that when you want to use the enum indexes their position in the type declaration is relevant: The first constructor declared has index zero and the index value is incremented by one for each subsequent constructor.

**Summary**

So many topics have been touched in this chapter; you now have all the knowledge needed to start writing some great code. Among other things, so far you have learned:

- What classes are and how you can use them in your applications
- How to use inheritance to organize your objects and to promote code reuse
- How to use typedef to define new types and how to extend them
- How to implement your own collections using iterators
- How enums differ from any other languages and how they can be effectively used to lean your code

In the next chapter, you learn how to organize your classes in files and packages, how to create code documentation automatically, and how to use Unit Testing in haXe.
Organizing Your Code

So far, you have learned a lot about types, classes, OOP, and so on. This chapter will show you how to manage the code for better organization and efficiency. This means breaking the code in multiple files that you can reuse from project to project, writing comments that can produce documentation automatically, and creating a set of tests to check that everything is fine whenever you make corrections or you introduce new functionalities. In this chapter, you will learn about the following topics:

- Dividing the code into packages
- Importing types from packages
- Adopting libraries written by others
- Using resources
- Commenting the code to produce documentation
- Unit testing
- Writing test code

Building Reusable Code

Many type definitions can be coded in a single file. This can be useful as long as the declarations are few and short, but this rarely happens in a real project. The solution is to divide the code into atomic blocks that are easy to reach and identify. To this purpose, haXe implements a package system to keep the code well organized and easy to reuse. A package is like a bucket that can contain type definitions separated in individual files and/or sub-packages. Each definition name must be unique in the same package.

A big advantage of packages is that they help avoid conflicting names. More than often, it happens that classes with the same name, Point for example, are defined in the application and in an external library used in the project. If the two were declared in the same file or in the same package,
there would be a conflict and the compiler simply cannot proceed but if they exist in different packages, and thus they can be easily distinguished, the compiler can do its job without a glitch.

Packages in haXe resemble the Java model and to some extent the namespaces as implemented in .Net.

Using Packages

The naming convention for packages is the same that applies to variables (alphanumeric sequence starting with a lowercase letter) and their names must match the directories that contain the package files; a package file normally has the same name of the main type defined in it with the conventional haXe extension `.hx`. When files and directories are named after types and packages, the compiler can do its job and easily find the position of each type definition.

When the haXe compiler is invoked, it is always necessary to point to an entry type definition, and if the corresponding file is encountered (type name + `.hx`), all the dependencies are resolved and the output produced. The compiler must have an entry type because it will not blindly assemble all the files in the current directory in the final output; it will just use what is really needed and discard everything else. This very smart feature allows the smallest possible footprint for output. Some dependencies are resolved automatically while others must be declared explicitly using the `import` statement, but first you will see how a package is declared.

For convenience a directory named `project` is declared and will contain everything that is needed to build the examples. The directory may be located anywhere on your file system. The `project` directory is considered the default package (a package without a name); haXe files that reside directly in the `project` directory should not declare a package. In the example, two directories are added to the directory corresponding to two packages with the same name, and some files are created in them:

```
project
  geom
    Circle.hx
    Point.hx
    // ...
  style
    Color.hx
    FillKind.hx
    // ...
  Main.hx
  GeomImport.hx
  // ...
```

Declaring a Package

The `package` statement must be the first in a file; any subsequent declaration will be automatically associated to it. A package is declared like this:

```
package name.subname;
```

Where `name` is the main package and `subname` is a nested package. The number of nesting levels is left to the programmers and his or her style of organizing the code.
The `geom` package contains a `Point.hx` file that declares the `Point` class, as in the following code:

```haxe
// content of file geom/Point.hx
package geom;
class Point
{
    public var x : Float;
    public var y : Float;
    public function new (x : Float, y : Float)
    {
        this.x = x;
        this.y = y;
    }
}
```

The complete name of the class `Point` is `geom.Point` while its name in the package is simply `Point`. The first line indicating the current package is mandatory and cannot be omitted.

*In this example and others in this chapter, classes are used in the examples, but in some circumstances and varying from project to project, other structures like typedef or enum may constitute better alternatives.*

### Implicit Import

The main types — their name matches the name of the file with `.hx` extension — can be referenced implicitly. It is possible to recur to a main type simply by using its complete name or the relative one if it is used in the same package.

The `Circle` class uses the `Point` class to define a variable that stores the center point of circle:

```haxe
// content of file geom/Circle.hx
package geom;
class Circle
{
    public var center : Point;
    public var radius : Float;
    public function new(center : Point, radius : Float)
    {
        this.center = center;
        this.radius = radius;
    }
    public function perimeter()
    {
        return( 2 * Math.PI * radius );
    }
}
```

Note that the `Point` class is used in this file without the `geom` package prefix and without any import declaration. The `Point` class is automatically discovered and immediately accessible, because it is in the same package as `Circle`. `Point` could also be referenced using its complete name, like this:

```haxe
public var center : geom.Point;
```
Part I: The Core Language

While using the complete name is optional when working in the same package, it becomes mandatory when referencing a type in a different package or in a sub-package.

```haxe
// content of file Main.hx
class Main
{
    static function main()
    {
        var p = new geom.Point(5, 5);
        var c = new geom.Circle(p, 10);
        trace(c.perimeter());
    }
}
```

The example file (Main.hx) is in the root directory of project and is the entry class used by the compiler. Because no package is declared, the class is part of the default empty package. When the previous example is compiled, the output will contain the definition of the three classes mentioned, because all three are used in the project; even if Point is not directly invoked by Main, Circle needs its presence to work properly.

**Explicit Import**

Another way to reference definitions on different files is to explicitly state that a certain file is required. This has two major effects: package names are no longer needed and now it is possible to reference all the types declared in the imported file and not just the main one. The import statements are introduced at the same level of the type and package definitions and can be positioned anywhere in the file but after the package declaration, if such exists. The import action will only affect the code written after the statement. The import declaration has this syntax:

```haxe
import package.Type;
```

Where package is the name of the package that contains the desired type and Type is the name of the file to import without extension; usually the filename corresponds to the main type defined in the file. Of course, the package name can be omitted when importing types from the default empty package.

As many import statements may be included in the same file as they are needed. Note that it is not possible to import whole packages but just one file at a time, importing the main type of each one.

Consider extending the Point class to Point3D. Usually this is done in a new file, but for the sake of explanation, the definition will be added in the Point.hx file.

It is not possible to reference implicitly the Point3D class from outside the Point.hx file, because it is not the main type of the file. To use this class, it is necessary to import all the type definitions included in the Point.hx file using the import statement.

```haxe
// content of file Main.hx
import geom.Point;
class Main
{
    static function main()
```
The complete name `geom.Point3D` can also be used. Complete names are convenient to disambiguate when the same type name is present in different packages and they are both used in the same file context.

**Enum and Packages**

Enum constructors behave exactly as the other types with the exception that in a switch statement, it is possible to use just the constructor name in place of the complete one, if the type is known at compile time and even if the `enum` file has not been imported.

```haxe
// content of file style/FillKind.hx
package style;
enum FillKind
{
  Solid(rgb : Color);
  Gradient(startColor : Color, endColor : Color);
  Texture(fileName : String);
}
```

The `enum` in the preceding code can be used in the `Main.hx` as follows:

```haxe
// content of file Main.hx
import style.FillKind;
class Main
{
  static function main()
  {
    var c = Texture("filename.jpg");
    // removing the import statement, the above line should be
    // var c = style.FillKind.Texture("filename.jpg");
    switch(c)
    {
      // with or without the import statement, the constructor
      // name can be used without indicating the complete name
      // as long as the "c" type is known
      case Solid(color):
        // ...
      case Gradient(startColor, endColor):
        // ...
      case Texture(file):
        // ...
    }
  }
}
```

The `Color` type can be an `enum`, a `typedef`, or a `class` and its implementation has been omitted because it is not important to understand the import logic.
Part I: The Core Language

**Type Lookup Sequence**

When the compiler parses a file and encounters a variable type, an argument type, or a return type, it will perform the search for its definition in the following order of priority:

- Type parameters like class parameters or `enum` parameters. It's not really a good practice to define a type parameter with the same name as an existing type, but it can happen incidentally when an external library is imported.
- Types like `Float` or `String`, because they are contained in the `StdTypes` module that is imported by default and contains the definition the standard types.
- Types declared in the current file.
- Types declared in the imported files.
- It tries to load a file matching the type name and it looks into it.

**Importing an Entire Package**

As already seen, it is not possible to import an entire package at once. To overcome this limit, it is possible to use a sort of shortcut file. The file will contain the import statements and a small `typedef` declaration for each type in the package, or just the ones that are frequently used together. Then when the whole collection of types is needed, an import statement is used to call the shortcut file.

```haxe
// content of file GeomImport.hx
import geom.Point; // the import is needed otherwise Point3D cannot be referenced
// ...
typedef Circle = geom.Circle
typedef Point = geom.Point
typedef Point3D = geom.Point3D
// ...
```

Which is used like this:

```haxe
// content of file Main.hx
import GeomImport;
class Main
{
    static function main()
    {
        var p1 : Point = new Point(1,2);
        // ...
        var p2 : Point3D = new Point3D(1,2,3);
        // ...
        var c : Circle = new Circle(p1, 10);
        // ...
    }
}
```

The `Point3D` implementation is not shown in the example but it is a class that extends `Point` and that resides in the same `hx` file. It is not the main type of the file and, thus, must be imported explicitly or indirectly as described previously.
Chapter 6: Organizing Your Code

This technique seems to load the output file with an extra burden for the newly introduced typedefs; in practice the compiled result will be exactly the same as using direct imports. This happens because typedefs are declarations used only to check the types at compile time and they are not actually translated into bytecode in the generated output.

In some cases, it is useful to create instances of classes using their names and the reflection API. It sounds obvious that no type check is done over the generated instances because they can only be identified at run time. If those classes are used only with reflection, the compiler cannot know that they are actually needed and will not include them in the output file. To make the things right, the class has to explicitly be imported somewhere in the code with the import statement.

Importing Libraries

Working with haXe, you will find that although the language is quite young, a lot of libraries and tools are already available. The haXe environment comes with a standard framework of classes that have a very generic purpose and more functionalities can be integrated using the haxelib tool.

haXe Standard Library

haXe distribution comes with a fully-fledged library with many useful tools. In the examples so far, some of those functionalities have been already explored. When using types defined in the standard library, it is enough to import the files needed or to reference their complete names; the compiler will treat them as part of the current project.

The code of the standard library can be found in the std directory of the haXe installation. It contains functionalities that are cross-platform and others that are platform specific. Table 6-1 gives a brief description of the included packages.

Table 6-1

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default (empty package)</td>
<td>Cross-platform (EReg is not supported in Flash version prior to 9). Contains core definitions for base types such as arrays, strings, dates, reflection, XML, and tools to deal with them.</td>
</tr>
<tr>
<td>flash</td>
<td>Applies to Flash for versions 6 to 8. The package doesn’t introduce new functionalities for the Flash player, but simply maps existing features to haXe-specific structures.</td>
</tr>
<tr>
<td>haxe</td>
<td>Cross-platform with some exceptions. Contains haXe-specific functionalities such as Remoting (communication between different layers written in haXe), unit testing, XML extended functionalities, templates, and so on.</td>
</tr>
</tbody>
</table>

Table continued on following page
Part I: The Core Language

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>js</td>
<td>Specific for JavaScript. Maps functionalities found in modern browsers to manage things like DOM (Document Object Model), cookies and XHR (XML HTTP Request).</td>
</tr>
<tr>
<td>neko</td>
<td>Specific for Neko. Maps existing functionalities of the underlying Neko Virtual Machine and adds some definitions for types used in server-side projects.</td>
</tr>
<tr>
<td>tools</td>
<td>Targets primarily Neko. Contains utility definitions to create documentations from haXe generated XML code and provides functionalities to manage haxelib libraries.</td>
</tr>
</tbody>
</table>

**haxelib Libraries**

Standard functionalities can be easily extended recurring to the haxelib. *haxelib* is both a concept, a repository of useful haXe extensions, and a tool, used to install and manage the extensions. The haxelib repository hosts several community-contributed projects that can be found at [http://lib.haxe.org/](http://lib.haxe.org/)

The number of projects and their updates is growing daily and at the time of this writing, the projects shown in Table 6-2 are available.

**Table 6-2**

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arctic</td>
<td>Lightweight GUI framework for haXe targeting Flash 8 and Flash 9.</td>
</tr>
<tr>
<td>aspell</td>
<td>Neko Aspell enables using GNU Aspell from Neko/haXe.</td>
</tr>
<tr>
<td>caurinaTweener</td>
<td>caurinaTweener is a powerful class that can create transitions for any numeric value of an object programmatically.</td>
</tr>
<tr>
<td>cptr</td>
<td>Neko module for handling plain C pointers.</td>
</tr>
<tr>
<td>dcxml</td>
<td>Alternative XML implementation for haXe.</td>
</tr>
<tr>
<td>fash</td>
<td>Cross-version methods targeting Flash 8 and Flash 9.</td>
</tr>
<tr>
<td>fcomponentshx</td>
<td>Set of components for haXe targeting Flash 8 and Flash 9.</td>
</tr>
<tr>
<td>fhx</td>
<td>Functional data structures for haXe.</td>
</tr>
<tr>
<td>flex</td>
<td>The header files (mx.* package) needed to manipulate Flex2 components from haXe.</td>
</tr>
<tr>
<td>fmod</td>
<td>Neko fmod enables using parts of the fmod audio library from Neko/haXe.</td>
</tr>
<tr>
<td>fonttools</td>
<td>Lightweight Neko interface to some FreeType and FontConfig functionality.</td>
</tr>
</tbody>
</table>
### Chapter 6: Organizing Your Code

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>framework_delfiweb</td>
<td>Application framework for Flash 9.</td>
</tr>
<tr>
<td>gtranslator</td>
<td>Google translator service.</td>
</tr>
<tr>
<td>haxealtdate</td>
<td>Library containing Timezones, Gregorian, and Julian calendar classes.</td>
</tr>
<tr>
<td>haxelib-test</td>
<td>This project is a test of a haxelib project.</td>
</tr>
<tr>
<td>haxORMap</td>
<td>A haXe Object Relational Mapping, SQL, and Class generator tool.</td>
</tr>
<tr>
<td>hxasm</td>
<td>A Flash9 assembler Library for Flash9 and Neko.</td>
</tr>
<tr>
<td>hxDev</td>
<td>IDE for haXe based on Geany.</td>
</tr>
<tr>
<td>hxDiff</td>
<td>Sequencer Matcher and Document Deltas.</td>
</tr>
<tr>
<td>hxGtk</td>
<td>GTK wrapper library.</td>
</tr>
<tr>
<td>hxJSON</td>
<td>JSON encoding and decoding.</td>
</tr>
<tr>
<td>hxLib</td>
<td>Haxelib hack, providing multiple repositories and server options, backended</td>
</tr>
<tr>
<td></td>
<td>by hxServlet.</td>
</tr>
<tr>
<td>hxScriptlet</td>
<td>JQuery style JavaScript library.</td>
</tr>
<tr>
<td>hxServlet</td>
<td>Fast Proxied haXe Servlets.</td>
</tr>
<tr>
<td>hxSynth</td>
<td>Synthesis of hxServlet and hxScriptlet.</td>
</tr>
<tr>
<td>lhx</td>
<td>Mathematical data structures.</td>
</tr>
<tr>
<td>Lsys</td>
<td>Functional data structures and L-System generators.</td>
</tr>
<tr>
<td>mtwin</td>
<td>Motion-twin public haXe libraries contains many tools that make a web</td>
</tr>
<tr>
<td></td>
<td>developer’s life easier.</td>
</tr>
<tr>
<td>nGame</td>
<td>nGame is an Allegro wrapper for the haXe and Neko platforms.</td>
</tr>
<tr>
<td>nMagick</td>
<td>nMagick is an ImageMagick wrapper for the haXe and Neko platforms,</td>
</tr>
<tr>
<td></td>
<td>providing unsurpassed image manipulation capabilities.</td>
</tr>
<tr>
<td>nme</td>
<td>NME stands for Neko Media Engine and is an SDL wrapper for the haXe and</td>
</tr>
<tr>
<td></td>
<td>Neko platforms.</td>
</tr>
<tr>
<td>nPostgres</td>
<td>nPostgres is a PostgreSQL adapter for the haXe and Neko platforms,</td>
</tr>
<tr>
<td></td>
<td>providing full SPOD capabilities.</td>
</tr>
<tr>
<td>opengl</td>
<td>Neko/haXe binding for OpenGL, GLU, and GLUT.</td>
</tr>
<tr>
<td>pascal4neko</td>
<td>With pascal4neko you can write neko .ndlls in object pascal (freepascal or</td>
</tr>
<tr>
<td></td>
<td>delphi), or to embed neko modules in pascal.</td>
</tr>
<tr>
<td>swhx</td>
<td>Screenweaver haXe enables the creation of Flash interfaced desktop</td>
</tr>
<tr>
<td></td>
<td>applications using haXe.</td>
</tr>
<tr>
<td>systools</td>
<td>A cross-platform extension to the Neko VM for accessing system APIs.</td>
</tr>
</tbody>
</table>

*Table continued on following page*
Part I: The Core Language

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>twister</td>
<td>An implementation of the Mersenne Twister pseudo-random number generator for haXe.</td>
</tr>
<tr>
<td>VectorSpace</td>
<td>This project contains some classes to work with Vectors.</td>
</tr>
<tr>
<td>xcross</td>
<td>xCross is a package to cross-build zero-depency haXe applications.</td>
</tr>
<tr>
<td>xinf</td>
<td>xinf is not flash: cross-runtime, cross-platform graphics.</td>
</tr>
<tr>
<td>xpath</td>
<td>Implementation of XPath for haXe.</td>
</tr>
</tbody>
</table>

Installing haxelib Libraries

Installing haxelib libraries is really simple. Open the command prompt (or terminal if on a non-windows machine) and type:

```
> haxelib install libname
```

Where `libname` is the name of the library you want to install. Note that the tool is case sensitive and you have to use the exact name as shown in the library repository. Once invoked the library is downloaded and uncompressed in the local library folder. Some libraries require a post installation action that can be performed using the following command.

```
> haxelib run libname
```

Note that on Windows Vista, a security dialog may appear when you use the haxelib tool; just confirm and go on. Table 6-3 lists all of the available switches for the haxelib command.

Table 6-3

<table>
<thead>
<tr>
<th>Command Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>config</td>
<td>Outputs the installation path of the libraries.</td>
</tr>
<tr>
<td>info projectname</td>
<td>Lists the history log for the specified project.</td>
</tr>
<tr>
<td>install projectname [version]</td>
<td>Downloads and installs the specified project. If no argument follows the project name it will be installed the latest version otherwise the specified one.</td>
</tr>
<tr>
<td>list</td>
<td>Lists all the installed projects and their versions. The versions in square brackets are the currently used.</td>
</tr>
<tr>
<td>path projectname</td>
<td>Outputs the directory path to the active version of the specified project.</td>
</tr>
</tbody>
</table>
Chapter 6: Organizing Your Code

<table>
<thead>
<tr>
<th>Command Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>Registers a new user in the haxelib repository. This is required to submit new projects to the community. To complete the registration, it will be required to provide the following information: username, e-mail address, full name, and password.</td>
</tr>
<tr>
<td>remove projectname [version]</td>
<td>Removed the specified project. If a version is indicated, only that will be removed, otherwise all the installed versions for the project will be removed.</td>
</tr>
<tr>
<td>run projectname [version] [args]</td>
<td>Executes a run script for the specified project; it applies to projects that require a post-installation process. The version argument is optional. Other arguments may be required by the library.</td>
</tr>
<tr>
<td>search keyword</td>
<td>Lists all the projects that match the keyword. To return the full list of available projects, use the wildcard %.</td>
</tr>
<tr>
<td>set projectname version</td>
<td>Activates a specific version of the indicated project. From now on, the default version used will be the activated one.</td>
</tr>
<tr>
<td>setup</td>
<td>Sets the path to the haxelib projects directory.</td>
</tr>
<tr>
<td>submit file.zip</td>
<td>Submits the indicated zip file to the haxelib repository on the haXe.org site. The zip file contains an entire library that you want to share with the community.</td>
</tr>
<tr>
<td>test file.zip</td>
<td>Installs the specified zip file into the local haxelib repository, useful to test projects before submitting.</td>
</tr>
<tr>
<td>upgrade</td>
<td>Upgrades all the installed projects to the latest version.</td>
</tr>
<tr>
<td>user username</td>
<td>Prints information on the user: id, username, e-mail, and projects maintained.</td>
</tr>
</tbody>
</table>

The haxelib tool also contains commands to submit custom projects to the community repository; you can find detailed instructions on how to submit your projects on the haXe.org website.

**Using the haxelib Tool**

Using the libraries in a project is very simple. Just add the following switch to your .hxml file or in the compiler command line:

```
-lib projectname[:version]
```

If no version is indicated, then the current one will be used. By default, the current version is the latest, unless a specific version has been installed or set as active using the set switch in haxelib command. When the source code is compiled this way, the compiler will treat the library as part of the project as it happens with the standard library.
Part I: The Core Language

**Libraries from Other Projects**

Many times a developer writes code that he or she wants to reuse in several projects. It is possible to add references from the current project to others using the switch `-cp` (class path) followed by the path to the directory containing the code to include. The `-cp` switch can be repeated more than once with different paths. The imported libraries will be treated as residents in the main directory of the current project.

Supposing that you have a library containing some useful classes in `d:\lib\utilities`, you can import them at compile time using the class path switch in the following way:

```
  haxe -cp d:\lib\utilities -neko out.n -main Main
```

This command compiles the `Main` class contained in the current working directory; the `Main` class can also access the types defined in the files contained in the utilities directory using the already explained import notation.

**External Libraries**

Sometimes it will be necessary to use third-party libraries that are not available in haXe code but just as compiled resources (in the JavaScript case a real compilation does not exist; haXe translates its own code into another syntax and adds a small API to even the differences between the two). When this is the situation, it is possible to use the compiled library but it will be necessary to use reflection and `untyped` blocks to reach the library contents. To simplify the access to the imported types, it is also possible to define some kind of header files (in the C/C++ sense) that map the library functionalities to classes usable in haXe.

**Extern Classes**

When a library that contains definitions for classes is imported and there is the need of a strictly typed approach, one or more `extern` classes must be defined. The haXe compiler, to check that the code is syntactically correct, will use those classes. The `extern` classes are different from normal classes because their functions do not implement a body; this is not required because the real implementation is encountered in the external library and the `extern` class works like a pointer to the external resource.

External classes can declare external functions for both static and instance fields, as shown in the following code;

```haxe
  extern class ExternalClass
  {
    public function externalInstanceMethod() : Void;
    public static function externalStaticMethod() : Void;
  }
```

In addition, `typedef` can be used to map external structures. This is the case for anonymous objects defined in the foreign library or also for classes as far as they are not instantiated using the `new` constructor.

Each platform deals with compiled external libraries in a very different way. The following sections will try to cover every possibility, even if some of them are very rare.
Flash External Libraries

There are two ways to link external libraries in the Flash platform: dynamically and statically. In the former solution, the external library is loaded at run time, while the latter uses the -swf-lib switch to embed the external library directly in the output file at compile time. The first solution makes it possible to load dynamically portions of code just when and if they are needed; the biggest disadvantage is that there is more code to write to manage the code loading and that things can go wrong if some synchronization fails. The second solution is easier to implement and will of course increase the size of the output binary.

To illustrate the inclusion of external libraries in Flash, a very simple class Drawer is defined in a new project directory; the class purpose is to draw simple shapes (a rectangle and a poly-line) on the passed movie clip (Flash 6 to 8) or sprite (Flash 9). It is important that the directory is different from the one of the current project or there is the risk that the class is erroneously included in the source compilation invalidating the examples. The class will be compiled in an swf file that will be included first in a static way and after in a dynamic way.

```
// content of file Drawer.hx
import flash.Lib;

class Drawer
{
  #if flash9
  private var g : flash.display.Graphics;
  public function new (mc : flash.display.Sprite)
  {
    g = mc.graphics;
  }
  #else flash
  private var g : flash.MovieClip;
  public function new (mc : flash.MovieClip)
  {
    g = mc;
  }
  #end

  public function drawRect(x : Float, y : Float, w : Float, h : Float) : Void
  {
    g.lineStyle(2, 0x000000);
    g.moveTo(x, y);
    g.lineTo(x+w, y);
    g.lineTo(x+w, y+h);
    g.lineTo(x, y+h);
    g.lineTo(x, y);
  }

  public function drawPoly(coords: Array<Array<Int>>)
  {
    g.lineStyle(2, 0x000000);
    g.moveTo(coords[0][0], coords[0][1]);
    for(i in 1...coords.length) {
      g.lineTo(coords[i][0], coords[i][1]);
    }
  }
}
```
Part I: The Core Language

Importing a library in the ways described later is insane when the source code is available and it can be used simply with the class-path switch. In the following examples you are simulating that the Drawer class is part of a foreign library that you need to use, but it is not available in source format or the source format is not compatible with haXe and you cannot translate it rapidly.

The class is targeting generically the Flash platform. Drawing functionalities, as almost everything else, have changed a lot between version 9 and the previous ones. In the previous code, the differences are removed using the conditional compilation. In this case, Flash 9 uses the Graphics class to draw, while versions 6 to 8 draw directly on a MovieClip instance.

To compile the above class targeting all the flash versions at once, an hxml file is created. The file must reside in the same directory as Drawer.hx.

```plaintext
# content of the file build-drawer.hxml
# Flash6
-swf f6.swf
-swf-version 6
Drawer
--next
# Flash7
-swf f7.swf
-swf-version 7
Drawer
--next
# Flash8
-swf f8.swf
-swf-version 8
Drawer
--next
# Flash9
-swf f9.swf
-swf-version 9
Drawer
```

The file can be processed using the following command from the command line:

```
> haxe build-drawer.hxml
```

This is equivalent to double-clicking on the file if the .hxml extension is properly associated to the haXe program in the operating system. In any case, the compiler will produce four swf files, one for each flash version.

A flash movie that loads another movie with a different version can generate unexpected results (more often it simply doesn’t work). As a general rule of thumb, a movie can load movies of the same versions; prior versions can be used but only to access their assets (images, sounds, and so on).

Static Link

Linking a file statically requires less code and is less error prone than the dynamic option because the external library is not loaded asynchronously but is embedded in the main file. The other advantage is that the external library will be available at the same time of the classes declared in the main project.
This simplified approach permits the use of the same exact code on all of the flash versions:

```haxe
// content of file Main.hx
import flash.Lib;
class Main
{
    static function main()
    {
        var cl = Type.resolveClass("Drawer");
        var drawer = Type.createInstance(cl, [Lib.current]);
        draw(drawer);
    }

    private static function draw(drawer : Dynamic)
    {
        drawer.drawRect(20, 20, 80, 100);
        drawer.drawPoly([[120, 10], [170, 100], [70, 100], [120, 10]]);
    }
}
```

The code is very easy to use and implement. The most important thing is to not forget to add the switch `-swf-lib` in the compiler command. The switch must be followed by the path of the file to import and it can be repeated to import more than one file. Flash 6 versions can be compiled in this way:

```
> haxe -swf index_f6.swf -swf-version 6 -swf-lib f6.swf -main Main
```

**The external libraries must be copied in the same directory as the source code to be reachable by the compiler, or the path in the switch statement must be changed to point the correct path.**

To enjoy the strictly typed constraint of the compiler, it is possible to define an `extern` class for the `Drawer` class.

```haxe
extern class Drawer
{
    #if flash9
        public function new (mc : flash.display.Sprite) : Void;
    #else flash
        public function new (mc : flash.MovieClip) : Void;
    #end
    public function drawRect(x:Float, y:Float, w:Float, h:Float) : Void;
    public function drawPoly(coords:Array<Array<Int>>) : Void;
}
```

The class can now be used as if the class is defined directly in the host project; see how the previous example can be simplified.

```haxe
// content of file Main.hx
class Main
{
    static function main()
    {
        var drawer = new Drawer(Lib.current);
        draw(drawer);
    }
} (continued)
```
Part I: The Core Language

(continued)

```haxe
private static function draw(drawer : Drawer)
{
    drawer.drawRect(20, 20, 80, 100);
    drawer.drawPoly([[120, 10], [170, 100], [70, 100], [120, 10]]);
}
```

Dynamic Link

In the dynamic link option, the main class loads dynamically the swf containing the `Drawer` definition and uses it to instantiate an object. The systems to load dynamically external movie clips in Flash vary a lot with each version. It is possible to write the code in a unique file using conditional compilation but for the sake of clarity, the example is repeated for Flash 6, Flash 7/8, and Flash 9.

```haxe
// content of file Main.hx for Flash 6
import flash.Lib;
import flash.MovieClip;
class Main
{
    public static function main()
    {
        var container = Lib.current.createEmptyMovieClip("container", 0);
        var loader = container.createEmptyMovieClip("loader", 0);
        container.onEnterFrame = function() {
            if(loader.Drawer != null) {
                container.onEnterFrame = null;
                var drawer = Type.createInstance(loader.Drawer, [Lib.current]);
                draw(drawer);
            }
        }
        loader.loadMovie("f6.swf");
    }
    private static function draw(drawer : Dynamic)
    {
        drawer.drawRect(20, 20, 80, 100);
        drawer.drawPoly([[120, 10], [170, 100], [70, 100], [120, 10]]);
    }
}
```

The external movie clip is loaded inside the main one using the `loadMovie` of the `MovieClip` class. To be sure that the imported class is available, the movie clip just loops in the `onEnterFrame` event waiting for the external movie to be completely loaded. Once loaded, the repeating call to `onEnterFrame` is removed, an instance of `Drawer` is created, and the function `draw` is invoked. Note that there is no information about the class `Drawer` at compilation time and, thus, the object must necessarily be passed as `Dynamic` in the `draw` function and the instance must be created using reflection. There are alternatives to the `onEnterFrame` event to ensure that the movie has been loaded (this example will end in an infinite loop if the `f6.swf` is not available for some reason and will not report any error about it), but this method is short and easy to read.
Chapter 6: Organizing Your Code

// content of file Main.hx for Flash 7/8
import flash.Lib;
import flash.MovieClip;
import flash.MovieClipLoader;
class Main {
    public static function main()
    {
        #if flash7
            var movie = "f7.swf";
        #else flash8
            var movie = "f8.swf";
        #end
        var loader = Lib.current.createEmptyMovieClip("loader", 0);
        var ml = new MovieClipLoader();
        ml.onLoadInit = function(mc : MovieClip) : Void
        {
            var drawer = Type.createInstance(mc.Drawer, [Lib.current]);
            draw(drawer);
        }
        ml.loadClip(movie, loader);
    }
    private static function draw(drawer : Dynamic)
    {
        drawer.drawRect(20, 20, 80, 100);
        drawer.drawPoly([[120, 10], [170, 100], [70, 100], [120, 10]]);
    }
}

The version 7 and 8 are functionally equivalent in this case; conditional compilation is just used to select the corresponding file for each version. The MovieClipLoader permits a much cleaner code syntax, and managing errors can be easily introduced adding a declaration for the ml.onLoadError function.

The draw function works the same as for version 6:

// content of file Main.hx for Flash 9
import flash.Lib;
import flash.display.MovieClip;
import flash.display.Loader;
import flash.events.Event;
class Main {
    static var c : Int;
    public static function main()
    {
        var loader = new Loader();
        loader.contentLoaderInfo.addEventListener(Event.COMPLETE, function(event : Event) : Void
        {
            var cl = event.currentTarget.applicationDomain.getDefinition("Drawer");
            var drawer = Type.createInstance(cl, [Lib.current]);
            draw(drawer);
        });
    }
}(continued)
Part I: The Core Language

(continued)

```actionscript
loader.load(new flash.net.URLRequest("f9.swf"));
flash.Lib.current.addChild(loader);
}
private static function draw(drawer : Dynamic)
{
    drawer.drawRect(20, 20, 80, 100);
    drawer.drawPoly([[120, 10], [170, 100], [70, 100], [120, 10]]);
}
}
```

The Flash 9 platform has changed a lot from the previous versions; functionalities have been broken apart into many smaller classes, and packaged in areas that are more specific. The code uses many more classes and instances but it is easier to read and debug. In this example, a Loader instance is used to load the external library, and the URL to the external movie clip must be wrapped in a URLRequest object. The Drawer definition is not globally accessible and must be retrieved from the loaded file context using the getDefinition() method; the rest of the code is very similar to the previous examples.

*Before running any of the preceding examples, remember to copy the previously compiled fversion.swf to the output directory otherwise the movie will not be able to find and load the library.*

*Note that the current implementation of haXe only allows the use of one -swf-lib switch at the time; you cannot actually embed more than one swf file.*

### Neko External Libraries

Neko applications can import two kinds of libraries: Neko bytecode files (.n) or libraries written in C and compiled into .ndll files to target the Neko VM (Neko Virtual Machine).

### Neko Bytecode .n

Neko bytecode is generated using the Neko run time. The code written in haXe is effectively converted into a file written in the Neko programming language (conventionally associated to the extension .neko), and then compiled by neko.exe in an .n file. It is possible to see what a .neko file looks like, adding the switch --neko-source, which preserves the intermediate generated file. Any .n file is a module that can contain an entire application or just a bunch of classes to use as a library. Usually the best bet is to merge the main project and the external library using the class path switch (-cp), but there are cases where this is not possible.

To illustrate the use of an external library dynamically loaded, one class External with one function say() is created.

```actionscript
class External
{
    public function new() { }
    public function say() : String
    {
        return "Hello world!";
    }
}
```

156
The class is then compiled using the following command:

```haxe
> haxe -neko external.n External
```

Because it does not make any sense to use this module alone, the `-main` switch has been omitted.

The external library is now ready to be loaded dynamically and to be used through reflection. Remember to put the generated `external.n` file in the same output directory as the `example.n` file before executing it.

```haxe
class Main
{
    static function main()
    {
        var loader = neko.vm.Loader.local();
        var module = loader.loadModule("external");
        var classes : Dynamic = module.exportsTable().__classes;
        var ob = Type.createInstance(classes.External, []);
        trace(ob.say());
    }
}
```

The module can be compiled with the following command:

```haxe
> haxe -neko example.n -main Main
```

To execute the code in the command line, use the following command:

```haxe
> neko example.n
```

The output will be:

```
Main.hs:7: Hello world!
```

Some explanations are required. The `neko.vm.Loader` is a Neko-specific class that permits the loading and management of external libraries. The static function `local` returns the `Loader` that was used to load the module in which the code is defined. The `loadModule` loads an external library defined in an `.n` file, the extension is automatically appended and the file will be searched in one of the directories accessible by the loader. The current directory and the installation directory of Neko are the default path where modules will be searched. More directories can be added using the `addPath` function. A loaded module is always cached for performance reasons. The cache system uses the name of the loaded module to identify it and it is possible to load a module more than once if different paths are provided.

The loaded module contains the class definitions that can be accessed using the reflection abilities of haXe. The `External` class could be contained in a package or sub-package; in that case the syntax to create an instance of it would be:

```haxe
var ob = Type.createInstance(classes.myPackage.mySubPackage.External, []);
```
Part I: The Core Language

**Neko .ndll**

Neko .ndlls are shared libraries written in C that target the Neko platform (see Table 6-4). Essentially, they are a bridge between the Neko run time and the C world. Writing an .ndll permits to wrap the functionalities defined in some low-level libraries with direct access to the operating system. This happens with .ndll that comes with the distribution and that can be found in the Neko installation directory.

<table>
<thead>
<tr>
<th>ndll Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod_neko2.ndll</td>
<td>Communicates with the Apache 2.2 web server.</td>
</tr>
<tr>
<td>mod_neko.ndll</td>
<td>Communicates with the Apache 1.3 web server.</td>
</tr>
<tr>
<td>mysql.ndll</td>
<td>Wraps the functionalities of the MySQL engine (the database must be installed independently).</td>
</tr>
<tr>
<td>regexp.ndll</td>
<td>Performs regular expression operations.</td>
</tr>
<tr>
<td>sqlite.ndll</td>
<td>Wraps the functionalities of the SQLite engine, but also embeds the original C library and; thus, no installation is required.</td>
</tr>
<tr>
<td>std.ndll</td>
<td>Contains functionalities to access to the filesystem.</td>
</tr>
<tr>
<td>ui.ndll</td>
<td>Contains functionalities to access user-interface related features on the support operative systems.</td>
</tr>
<tr>
<td>zlib.ndll</td>
<td>Manages compression and decompression of archives.</td>
</tr>
</tbody>
</table>

Even if the Neko VM is extremely fast in executing code, there are some rare circumstances where a better optimization can be obtained using C libraries.

In Neko a primitive is a glue function that is used to bridge the C code into Neko. Primitives can be accessed from haXe using the **Loader** class.

Suppose you have a **helloworld.ndll** that declares a function this way:

```
#include <neko.h>

value say()
{
    return alloc_string("Hello world");
}
DEFINE_PRIM(say, 0);
```

In haXe it will possible to use the preceding function like this:

```
var say : Void -> Void = neko.Lib.load("helloworld", "say",0);
trace(say());
```

The **loadPrimitive** will pass a reference to the function **say** in the C library. The second parameter 0 is the number of arguments that the function requires. You can find more information about C integration in the third part of the book and especially in Chapter 20.
Chapter 6: Organizing Your Code

**JavaScript External Libraries**

The community of web developers has produced plenty of libraries in JavaScript from novice code to very interesting frameworks and applications. Having the possibility to reuse this code without reinventing the wheel can be a real timesaver. External libraries are neither dynamically loaded (although possible) nor embedded (possible, too, but very clumsy). To be accessible the libraries must only be included in the same HTML page where the haXe code is intended to be used.

To illustrate the use of an external library in JavaScript a simple class `Panel` is defined in the `external.js` file. The class accepts the name of an element (attribute id in the HTML element) as argument for its constructor and is able to perform some simple actions on the element such as dragging it and switching its visibility.

```javascript
// content of file external.js
function Panel(name)
{
    var el = document.getElementById(name);
    this.el = el;
    el.style.left = el.style.left || el.offsetLeft;
    el.style.top = el.style.top || el.offsetTop;
    this.visible = true;
    this.draggable = false;
}
Panel.prototype.activateDrag = function()
{
    var el = this.el;
    el.style.backgroundColor = "#eee eee";
    el.onmousedown = function(e) {
        e = e || event; // normalize event
        if(!e) return; // no compatible browser
        var pos = {
            x : parseInt(el.style.left),
            y : parseInt(el.style.top),
            h : e.clientX,
            v : e.clientY
        }
        document.onmouseup = function(e)
        {
            document.onmousemove = null;
        }
        document.onmousemove = function(e)
        {
            if(!e) e = event; // normalize event
            el.style.left = e.clientX + pos.x - pos.h + 'px';
            el.style.top = e.clientY + pos.y - pos.v + 'px';
            return false;
        }
    };
    this.draggable = true;
}
Panel.prototype.deactivateDrag = function()
{
    this.el.onmousedown = null;
    this.el.style.backgroundColor = "#cccccc";
}
```

(continued)
Part I: The Core Language

(continued)

this.draggable = false;
}
Panel.prototype.hide = function()
{
  this.visible = false;
  this.el.style.display = 'none';
}
Panel.prototype.show = function()
{
  this.el.style.display = 'block';
  this.visible = true;
}

JavaScript is a programming language that does not include a formal definition for classes, but a very similar structure can be obtained using functions as objects and prototypes.

A wrapper extern class is defined, so that access to the previous code in haXe will be strictly typed and friendly.

// content of file Panel.hx
extern class Panel
{
  public function new(name : String) : Void;
  public function activateDrag() : Void;
  public function deactivateDrag() : Void;
  public function hide() : Void;
  public function show() : Void;
  public var visible(default, null) : Bool;
  public var draggable(default, null) : Bool;
}

The Panel class is used in the example to control the status of a box on the web page. The page also includes two buttons, one to pause and to resume the drag functionality of the box, the other to toggle its visibility. Those behaviors are wired in the main function.

// content of file Main.hx
import js.Dom;
import js.Lib;
class Main
{
  static function main()
  {
    var panel : Panel = new Panel("mypanel");
    panel.activateDrag();
    var btnv : Button = cast Lib.document.getElementById("buttonVisibility");
    btnv.onclick = function(event : Event)
    {
      if(panel.visible)
      {
        panel.hide();
      }
Chapter 6: Organizing Your Code

    btnv.value = "show";
    ) else {
        panel.show();
        btnv.value = "hide";
    }
    }
    var btnd : Button = cast Lib.document.getElementById("buttonDrag");
    btnd.onclick = function(event : Event)
    {
        if(panel.draggable)
        {
            panel.deactivateDrag();
            btnd.value = "start drag";
        } else {
            panel.activateDrag();
            btnd.value = "pause drag";
        }
    }
}

Compiling the preceding code with the following command produces a main.js script that can be included in the test page.

    > haxe -js main.js -main Main.hx

In the main function, a Panel instance is created and used just as if it were defined in the haXe project. Then a reference to the two buttons is created and used to add the behaviors to their onclick events. The object document represents the entire page content and has a method getElementsByTagName(name) that looks for a single element in the page with the attribute id equal to name. Because this function always returns an object of type js.Dom, it is necessary to cast it to the proper typedef Button. Note that the majority of the structures in the js package is mapped as a typedef and not as a class.

The sample page is very simple. It defines a style for the box, two buttons, the box to use with the Panel class and finally the inclusion of the external library and the haXe application. It is important to note that the JavaScript code will act on the structure of the HTML and to be sure that the DOM (Document Object Model) is ready to be modified, the .js files are included as the last elements. Other techniques exist to maintain the script tags in the head element and guarantee that the code is executed at the right time and they always leverage on some onLoad event that is browser specific.

    <html>
    <head>
        <title>External Libraries with Javascript</title>
        <style>
#mypanel {
    display: block; font-family: Verdana; font-size: 0.8em; position: absolute;
    width: 120px; height: 120px; border: 2px solid #999999; left: 180px;
}
    </style>
    </head>
    <body>

    (continued)
As good practices require, the presentation layer and the logic layer are completely separated; one of the advantages of this separation is that it is easier for both the designers and the programmers to deal just with what they do best.

Opening the web page index.html in a web browser produces the effect shown in Figure 6-1; using the left mouse button pressed when the cursor is over the gray box, the box can be dragged on the page; the two buttons on the left are used to pause and resume the dragging functionality and to hide and show the box.
Using Resources

The haXe compiler allows you to embed in the compiled output the content of files that are not code. They can be simple text, localization files, or configuration files. To include a resource, it is necessary to add the switch `-resource` as follows:

```
-resource resourcefile@resourcename
```

Where `resourcefile` is the path to the file to embed and `resourcename` is an identifier that will be used in the code to reference the resource. The static function `resource` of the `Std` class is used to access the content of the resource in this way:

```
var content : String = Std.resource("resourcename");
```

As an example, a `Config` class is defined that will recur on an embedded XML file to store configuration information about the database connections. More than one connection can be defined in the XML and the configuration can contain parameters for both MySQL and SQLite engines. The configuration class returns a default connection configuration when the `getDatabase()` method is used without the `name` argument. The method returns an anonymous object that contains the information defined in the XML file.

In the following example, two connections are configured but one or more can be used. Although no formal validation is done in the class `Config`, the system will throw very specific errors if some important information is missing or it will assume some default values when appropriate.

```
<config>
  <db>
    <connection name="primary" type="mysql" default="true">
      <mysql>
        host="localhost"
        user="primaryuser"
        pass="mysecretpassword"
        database="MyDb"
      </mysql>
    </connection>
    <connection name="secondary" type="sqlite">
      <sqlite>
        file="local.db"
      </sqlite>
    </connection>
  </db>
</config>
```

The compilation command (or the `hxml`) is modified to include the XML file:

```
> ... -resource config.xml@config_xml
```

The file `config.xml` is now embedded in the compiled output and can be accessed in code using the identifier `config_xml`.

The parameters for MySQL and SQLite are not the same; to account for those differences they are stored in a variable of type `Dynamic`. They can be cast to their relative `typedef` to get access to them in a strictly typed manner.
Part I: The Core Language

// content of file Config.hx
typedef MySqlParams = {
    database : String,
    host : String,
    port : Int,
    user : String,
    pass : String,
    socket : String
}
typedef SqliteParams = String
typedef DbInfo = {
    name : String,
    type : String,
    isdefault : Bool,
    params : Dynamic
}
class Config
{
    private static var xml = Xml.parse(Std.resource("config_xml")).firstChild();
    private static var dbElement = xml.elementsNamed("db").next();
    public static function getDatabase(?name : String) : DbInfo
    {
        // if no name is passed to the function, use the default db name
        if(name == null)
        {
            name = getDefaultDatabaseName();
        }
        for(el in dbElement.elementsNamed("connection"))
        {
            if(el.get("name") == name)
            {
                if(el.get("type") == "mysql")
                    return getMySqlDbInfo(el);
                else if(el.get("type") == "sqlite")
                    return getSqliteDbInfo(el);
                else
                    throw "Invalid db type " + el.nodeName + ";";
            }
        }
        return throw "No db section has the attribute name equal to " + name + ";";
    }
    private static function getDefaultDatabaseName() : String
    {
        if(dbElement == null)
            throw "There is no 'db' section in the xml configuration file";
        // look for a connection element with the attribute 'default'
        // set to 'true'
        for(el in dbElement.elementsNamed("connection"))
        {
            if((el.get("default") == "true")
            {
                return if(el.exists("name"))
            }
Chapter 6: Organizing Your Code

    el.get("name");
      else
        throw "The default db tag has no name attribute";
    }
  }
  // if no one is found, throw an error
  return throw "No default database is defined";
}
private static function getMySqlDbInfo(el : Xml) : DbInfo
{
  var p : Xml = el.elementsNamed("mysql").next();
  if(p == null)
    return throw "The MySql conf. " + el.get("name") + ", needs parameters";
  return {
    name : el.get("name"),
    type : "MySql",
    isdefault : el.get("default") == "true",
    params : {
      database :
        if(p.exists("database"))
            p.get("database")
        else
            throw "No database parameter in " + el.get("name") + ",",
      host : if(p.exists("host")) p.get("host") else "localhost",
      port : if(p.exists("port")) Std.parseInt(p.get("port")) else 3306,
      user : if(p.exists("user")) p.get("user") else "root",
      pass : if(p.exists("pass")) p.get("pass") else ",",
      socket : if(p.exists("socket")) p.get("socket") else null
    }
  }
}
private static function getSqliteDbInfo(el : Xml) : DbInfo
{
  var p : Xml = el.elementsNamed("sqlite").next();
  if(p == null)
    return throw "The Sqlite conf. " + el.get("name") + ", needs parameters";
  return {
    name : el.get("name"),
    type : "Sqlite",
    isdefault : el.get("default") == "true",
    params : {
      file : if(p.exists("file"))
        p.get("file")
      else
        throw "There is no database file for " + el.get("name") + ","
    }
  }
}
Part I: The Core Language

Assuming that the Config class is defined in the Config.hx file and that it resides in the default package, the configuration class can be used as follows:

```haxe
class Main
{
    static function main()
    {
        var dbInfo = Config.getDatabase();
        var conn = if(dbInfo.type == "MySql")
            neko.db.Mysql.connect(dbInfo.params);
        else if(dbInfo.type == "Sqlite")
            neko.db.Sqlite.open(dbInfo.params);
        else
            throw "Unsupported database '" + dbInfo + ":";
        // do something with the conn object
        // ...
    }
}
```

Because the Config class is used to open a database connection, it is obvious that this last example will work just on the Neko platform.

Documenting Your Code

Documentation is a critical component of development when developers work in teams; but it is also fundamental when a single developer wants to share his code with other developers and even in the situation that the code is not meant to be redistributed, it can be a real pain to get back to an undocumented code written a few months earlier. Luckily, haXe provides a nice tool to produce documentation automatically using the code itself and its comments.

The compiler when used with the flag `-xml` produces an XML file that contains the full description of all the code used in the compiled project. The XML contains information about every aspect of the code from class definitions to variable types. In addition, some of the comments are preserved and conveniently associated to the code they describe. To be included in the XML file, the comments must be of the block type and use two asterisks after the slash instead of a single one; all comments that do not conform to this signature will be discarded. As an example, the following `enum` has been correctly commented for documentation.

```haxe
// fragment of file QueryTools.hx
/**
 * A condition is a test to perform in the query. Many conditions can be
 * joined together using the "and" or "or" constructors.
 */
enum Condition
{
/**
 * Test if the passed field is Null.
 */
```
Chapter 6: Organizing Your Code

```haxe
TestIsNull(field : String);
/**
 * Test a numeric field against a value using a numeric operator.
 */
TestNumber(field : String, operator : NumericOp, value : Float);
/**
 * Test a text field against a text value using a text operator.
 */
TestText(field : String, operator : TextOp, value : String);
/**
 * Joins many conditions in a group. The group will evaluate to true if
 * all of its members evaluate to true.
 */
And(tests : Array<Condition>);
/**
 * Joins many conditions in a group. The group will evaluate to true if
 * at least one of its members evaluates to true.
 */
Or(tests : Array<Condition>);
}
```

When compiled with the following command, a `doc.xml` file is also produced.

```
> haxe -neko neko.n -xml doc.xml QueryTools
```

Now there are two options to produce a readable document: using the `haxedoc` tool to produce offline documentation or using the `haxedoc.n` bytecode to create online documentation.

**Offline Documentation**

When executed the haxedoc creates a directory `content` that will contain many HTML files — one for each type defined, one subdirectory for each package, and an `index.html` file that will contain a list of links to all the generated documents. Note that the documentation will contain information about every type imported in the application and those that are part of the standard libraries or the haxelib.

The `haxedoc` command accepts the following syntax:

```
haxedoc file.xml;platform -f ClassName
```

Where `file.xml` is the XML generated by the haXe compiler, `platform` is an indication that this file refers to a specific platform (multiple platforms can be used) and `-f` (filter) is used to produce documentation for only the specified type (the switch can be repeated to include more types). Both `file.xml` and the `-f` switch can repeated more than once with different values. In the first case, documentations from different platforms will be merged in a single structure to avoid duplication of information; repeating the filter switch permits you to produce documentation for multiple files.
Part I: The Core Language

Executing the following command in the directory containing the doc.xml produces a set of HTML files that can be explored using a standard browser.

```
> haxedoc doc.xml;neko
```

If the generated index.html page is opened in a web browser, a list of links will be presented. The aesthetic of the file is a little bit raw and this is because no template has been used to produce the documentation. To create a more appealing document, copy the file template.xml that is included in the directory std/tools/haxedoc of the haXe installation in the same folder of the XML files. This template is the same used for the API documentation on the site www.haxe.org. Otherwise, it is possible to use this template as a reference and to change its characteristics to fit the project needs. Figures 6-2 and 6-3 show the index page and the documentation of the enum described in the previous example with a custom template applied.

Figure 6-2
Chapter 6: Organizing Your Code

Online Documentation

The online version is used when the pages are to be accessed through an Internet connection. The pages are not statically generated but created on request. To use this tool, copy (a symlink is a better option if using a non-windows operating system) the file `haxedoc.n` from the directory `std/tools/haxedoc` into a folder that can be put online by your web server (using Apache with Neko is the topic of Chapter 9). The documentation XML files should be located in the relative path `../data`. The online version of haxedoc doesn't accept any configuration parameter, the XML files must be named after their platform (`flash.xml`, `javascript.xml`, `neko.xml`), and must be present to prevent errors. What was said about `template.xml` is still valid here.

Note that the links used in the documentation are in the format `/api/package/type`. This kind of path must be mapped to the format used internally by the `haxedoc.n` application. This format is as follows:

```
haxedoc.n?class=package/type
```
Part I: The Core Language

To obtain the effect, add the following content to the file .htaccess in the same directory or create a new one if it does not exist:

```
<FilesMatch "^([^_a-zA-Z0-9-]+)+$">
  RewriteEngine On
  RewriteRule /api/(.*) /haxedoc.n?class=$1
</FilesMatch>

This will only work on the Apache web server with the mod_rewrite module activated.
```

If you find the generator too restrictive or you want to tweak something, you can change or better extend the files contained in the std/tools/haxedoc. You will discover that the two haxedoc versions described so far are in reality exactly the same application with the only difference that the offline generator has been embedded in an .exe using the following command:

```
nekotools boot haxedoc.n
```

The file that contains the procedures to write the HTML documents is HtmlPrinter.hx. The whole document generation is provided using the RunTime Type Information (RTTI) facilities implemented in the haXe standard library. To know more about RTTI, consult Chapter 16.

Unit Testing

In information technology, Unit Testing is the practice to validate atomic portions of code; the smaller the portion the better it is. A unit in OOP usually refers to a class where at least one test is written for each of the class public methods. Nevertheless, what is a test in practice? A test is a single function that performs a single action and checks that the performed action has provided the expected results. If the results are correct, the test is passed; otherwise the test fails. Having unitary tests makes the developer’s life easier because when a bug is involuntarily introduced in the code during alterations, it is highly probable that one of the already written tests will fail indicating with a certain degree of precision where the error is. This works when the application code is covered by an extensive number of tests (a concept known as code coverage). When a bug is encountered and no tests are failing, the better thing to do is to write a new test that reproduces the exact same defect. In this way, it will be easier to check that the problem is not accidentally reintroduced in subsequent alterations.

Many developers consider Unit Testing a waste of time. This is because possibly they do not consider all the positive effects that a good test base has on their projects. Having a good code coverage means that alterations to the code can be made with confidence because if something critical is touched, a test will fail addressing the problem. Another good aspect of having tests is that they provide real examples for the developers that want to use the tested library; the code in the tests is easy to read because it is short for its own nature and provides a sort of living document. A positive side effect is that a developer used to writing tests is more concerned in writing code that has less dependencies and is more isolated from the context: this is because testing classes with a lot of dependencies can be a real pain.

The Unit Testing concept is not bound specifically to a technical implementation and potentially a Unit Test could also be a set of instructions manually written on a piece of paper; what renders the tests really effective is when they are aggregated in an automated context. The tests are aggregated in a processing unit that in one go executes all of them, reporting the number of tests run and in this case the number of
Chapter 6: Organizing Your Code

tests that have failed and their names. The automatic process is repeated many times during the development to help spot bugs and this is the reason why tests are often optimized for performance. Developers tend to drop testing when they are too slow to execute or getting them to work is too laborious.

The unitary testing is also in the execution context; each test must be independent from the others and must not rely on a specific configuration. If the tests are interconnected, having one that fails may compromise an entire chain of tests and render the bug individuation much harder. The context should not influence the tests because it is critical that tests can be moved with confidence from one machine to another and with the minimum of configuration possible.

There are areas where Unit Testing does not fit so easily, nominally UI (User Interfaces) and databases. This is because in the first case the user interaction is required, which tends to remove the automated advantage of the process. The databases are hard to test for miscellaneous facts: They must be configured, structures of tables must exist, and data for the tests must be in some way generated and deleted continually and be in the same exact state when the same test is repeated. The developer community provides many strategies to deal with these problems and going much deeper in Unit Testing techniques is beyond the scope of this book.

The haxe.unit Package

The standard library provides a very small but very useful package to write and run tests. The package contains just four classes, two of which are of interest to the programmer: TestCase and TestRunner.

**TestCase and TestRunner**

A test is a single function inside a class. The function must have a name that starts with the lowercase test keyword. This convention is used to distinguish between the methods that are really tests and the ones that are just helpers. The container class must extend the TestCase class. The class will inherit a very small set of functions (see Table 6-5) that will be used to write the tests.

**Table 6-5**

<table>
<thead>
<tr>
<th>TestCase Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup() : Void</td>
<td>This function is invoked automatically before each test is executed. In the TestCase class its implementation is empty; when overridden it must set the context so that the test can work as expected. This could be opening a needed database connection, reading a configuration file, and so on.</td>
</tr>
<tr>
<td>tearDown() : Void</td>
<td>This is complementary to the setup() function and is invoked after the test execution. Its main purpose is to cleanup what the test has left (deleting test records or files, and so on). The cleanup phase is critical to maintain a testing environment consistent each time the automation is repeated.</td>
</tr>
</tbody>
</table>

*Table continued on following page*
Part I: The Core Language

<table>
<thead>
<tr>
<th>TestCase Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertTrue(b:Bool, ?c:PosInfos) : Void</td>
<td>This is the core method for testing. Inside a test function, assumptions are made using the assert methods. If one of the assertions in the test is wrong, the test fails; otherwise the test passes. The b parameter must be true for the assertion being valid.</td>
</tr>
<tr>
<td>assertFalse(b:Bool, ?c:PosInfos) : Void</td>
<td>It’s the opposite of the assertTrue function; the assertion is successful when b is false.</td>
</tr>
<tr>
<td>assertEquals&lt;T&gt;({expected:T , actual:T, ?c:PosInfos}) : Void</td>
<td>This compares the expected value with the actual value and passes if the two are equal; otherwise it fails.</td>
</tr>
</tbody>
</table>

All the assert methods accept a last optional parameter of type PosInfos; as explained earlier, this is a special parameter that, when omitted, will be automatically fed by the compiler with an anonymous object containing information about the code context in that particular position.

The other important class of the package haxe.unit is the TestRunner. This class executes the tests that inherit from TestCase class, collects their results, and reports them. Each test class is added to a TestRunner instance using the add methods. They are collectively executed using the method run.

The haxe.unit targets any platform supported by haXe and the results will be reported in text format as much as the trace function does.

Writing Tests

Without going too deep into the topic (many books and sites discuss the argument), there exists a technique known as TDD (Test-Driven Development), which advocates that a good practice is to write the tests before the method implementations. The process is divided in the following steps:

- A test is written.
- The minimum of code required by the compiler to work is added in the class definitions. The code does not have to work in this phase. It is enough that the compiler can just produce an output.
- The test is run and will most likely fail because the model methods have not really been implemented.
- The code is changed to provide the minimum required code to make the test pass.
- The test is run and will pass.
- The code is refactored; if the change is introducing a bug, one of the tests will report the failure.

The process is then repeated until all the functionalities required by the application are implemented. Coding only the minimum required to pass the tests ensures that the application is not pretending to do
more than strictly necessary. This concept is resumed in the two acronyms KISS (Keep It Simple, Stupid) and YAGNI (You Ain’t Gonna Need It).

Although the size of each of the described steps is very small, it must be realistically dimensioned on the capacity and the personal attitudes of the programmer.

In the following example, this technique will be adopted in just the first steps and with a certain degree of freedom to not bore to the reader. It will be implemented in the base for a simple validation framework. What is needed is a very simple object that performs a validation and returns true if the validation is OK or false if it is not; in the latter case an error message will be produced.

So let us start with the first test case class. TestERegValidator extends TestCase. Usually the test case classes are named after the class name that they pretend to test. The first validation class will accept a text value and a regular expression pattern to perform the validation.

```haxe
class TestERegValidator extends haxe.unit.TestCase
{
    public function testValidateFalse()
    {
        var v = new ERegValidator("test", "x", null);
        assertFalse(v.validate());
    }
}
```

Remember that regular expressions are not present in Flash prior to version 9.

In the test an instance of the class ERegValidator is instantiated. The first test will check that a string value test does not match the pattern x; this is expressed using the assertFalse function that will fail if the passed argument is true if it is not; the test case class and the first test are ready but it is evident that this code alone cannot be compiled because the class ERegValidator doesn’t exist yet.

```haxe
class ERegValidator
{
    public function new(value : String, pattern : String, opt : String) {}
    public function validate() : Bool
    {
        return true;
    }
}
```

Now the code is ready to be compiled; but one last thing is missing, an entry class and a main method to run the tests and see the results.

```haxe
class Main
{
    static function main()
    {
        var runner = new haxe.unit.TestRunner();
        runner.add(new TestERegValidator());
        runner.run();
    }
}
```
Part I: The Core Language

To make this code work on all of the three platforms, the following `.hxml` file is used for compilation. Having the `-debug` switch activated when compiling unit tests is particularly critical because in case of failures it is easier to spot and to correct the errors.

```cpp
# Neko
-neko main.n
-main Main
--debug
--next
# Flash9
-swf main.swf
-swf-version 9
-main Main
--debug
--next
# Javascript
-js main.js
-main Main
--debug
```

Once compiled the result will be the same on every platform:

```cpp
Class: TestERegValidator F
* TestERegValidator::testValidateFalse()
ERR: Main.hx:12(TestERegValidator.testValidate) - expected false but was true
// here are omitted some lines of debugging information
// that vary with the different platforms
FAILED 1 tests, 1 failed, 0 success
```

This was expected; the code compiles but does not work because the validation function always returns `true` and the assertion expects a `false` argument to pass. Therefore, only the code that is needed for the test to pass is introduced:

```cpp
class ERegValidator
{
    public function new(value:String, pattern:String, ?opt:String) {}  
    public function validate() : Bool  
    {
        return false;
    }
}
```

Now the execution of the code will report:

```cpp
Class: TestERegValidator .
OK 1 tests, 0 failed, 1 success
```
A new failing test is introduced:

```java
public function testValidateTrue()
{
    var v = new ERegValidator("test", "t", null);
    assertTrue(v.validate());
}
```

This time the tests run two and one of them fails. It is time to accommodate the code so that both the tests validated may pass.

```java
class ERegValidator
{
    private var value : String;
    private var pattern : String;
    private var opt : String;

    public function new(value : String, pattern : String, ?opt : String)
    {
        this.value = value;
        this.pattern = pattern;
        this.opt = if(opt == null) "" else opt;
    }

    public function validate() : Bool
    {
        var er = new EReg(pattern, opt);
        return er.match(value);
    }
}
```

The constructor parameters are now maintained in the instance and effectively used for the validation. Now, in case of failure, an error must be available to inform the user that some problems prevent the validation process to be successfully completed. A new test is added:

```java
public function testEmptyError()
{
    var v = new ERegValidator("test", "t");
    v.validate();
    assertEquals(v.error, null);
}
```

A new variable error is required in the class and in the case of successful validation, it must have a null value.

```java
public var error(default, null) : String;
```

Another test has been addressed.

The process of creating tests and implementing code is then repeated many times until all the functionalities desired have been implemented. The result of all those iterations is the completion of the class ERegValidator; a new class EmailValidator has also been introduced with its complementary test case class TestEmailValidator.
class TestEmailValidator extends haxe.unit.TestCase
{
    public function testConventional()
    {
        var v = new EmailValidator("john@example.com");
        assertTrue(v.validate());
    }
    public function testDirty()
    {
        var v = new EmailValidator("john @example.com"); // spaces are not allowed
        assertFalse(v.validate());
    }
    public function testIncomplete()
    {
        var v = new EmailValidator("john");
        assertFalse(v.validate());
    }
    public function testDoubleDotted()
    {
        var v = new EmailValidator("john@example..com");
        assertFalse(v.validate());
    }
}
class TestERegValidator extends haxe.unit.TestCase
{
    public function testValidateTrue()
    {
        var v = new ERegValidator("test", "t");
        assertTrue(v.validate());
    }
    public function testValidateFalse()
    {
        var v = new ERegValidator("test", "x");
        assertFalse(v.validate());
    }
    public function testEmptyError()
    {
        var v = new ERegValidator("test", "t");
        v.validate();
        assertTrue(v.error == null);
    }
    public function testNotEmptyError()
    {
        var v = new ERegValidator("test", "x");
        v.validate();
        assertTrue(v.error != null);
    }
    public function testErrorContent()
    {
        var value = "test";
        var pattern = "x";
        var v = new ERegValidator(value, pattern);
        v.validate();
        assertTrue(v.error.indexOf(value) >= 0);
        assertTrue(v.error.indexOf(pattern) >= 0);
    }
}
class EmailValidator extends ERegValidator
{
    public function new (email : String)
    {
        super(email, "^[^@\s]+@((?:[-a-z0-9]+\.)+[a-z]{2,})$", "i");
    }
}

class ERegValidator {
    public var error(default, null) : String;
    private var value : String;
    private var pattern : String;
    private var opt : String;
    public function new(value : String, pattern : String, ?opt : String)
    {
        this.value = value;
        this.pattern = pattern;
        this.opt = if(opt == null) "" else opt;
    }
    public function validate() : Bool
    {
        var er = new EReg(pattern, opt);
        if(er.match(value))
            return true;
        else {
            error = "'"+value+"' does not match the expression /"+pattern+"/";
            return false;
        }
    }
}

The framework can be extended and completed adding new validation classes for numeric ranges, credit card numbers, phone numbers, and so on. A common interface Validator can be introduced to make the classes exchangeable and maybe changing the error variable type from String to List<String> could be a good idea. This way a single Validator could notify more than one reason of failure. A generic ValidatorGroup class could be introduced to perform many chained validations at once.

Summary

In this chapter you learned a lot about structuring the code so that it will make your work easier. A lot was covered and you now can master the following arguments:

- Packages
- Using and producing external libraries
- Documenting the code using the XML files generated by the compiler and the comments in the code
- Unit Testing and how to write them

In the next chapter, you learn how to deal with error handling and debugging in general. It will complete the first part of the book and you will acquire the basis to write real-world applications.
Okay, let’s face it; all programmers make mistakes. In fact, most programmers make lots of mistakes, and it seems the more experienced you are, the more likely you’re going to wind up pulling out tufts of hair over the most stupid mistakes imaginable. Now, while a bald head will make the women believe you’re a virile stallion, or at least save on lighting bills in the office, surely you’d rather save on the price of a toupee for the time being and just get the darn code working.

Every programmer has their own preference of a debugging toolkit for their favorite language. Most languages even come with a proprietary debugger and output panel set nicely integrated into the associated IDE. However, with haXe in its infancy, it doesn’t even come with its own IDE. Yet! Despite this, haXe does still provide quite a substantial array of features oriented to debugging the frilly bits out of your buggy code. What’s more, each feature is well thought out, powerful and useful, leaving you wondering why the platform creators hadn’t supplied as much exception and logging functionality themselves.

In this chapter, you will discover:

- How to trace your code for each platform
- How to extend the tracing capability
- How to reroute the existing trace functionality to an alternative handling mechanism
- What purpose exceptions facilitate in haXe
- How to handle exceptions in your classes

**The Trace Function**

Those of you who are familiar with programming for ActionScript will already know the `trace` function and all it entails. This neat little function allows the programmer to parse data to an output panel in the Flash IDE, providing a large percentage of the debugging capabilities used by...
Part I: The Core Language

Flash developers. The same is now true of haXe developers. How can this be done without an IDE? The answer is simple. Instead of parsing the passed string data to an output panel, it is instead displayed in a scrollable text field that is overlaid on top of the rest of the Flash movie. You are no longer bound to using clunky output apps in order to view content spouting through the Flash trace function. What’s more, this cool little function is supported by the JS and Neko platforms, too. Of course, these platforms display things a little differently. Though, you already knew all of this. If you recall back in Chapter 2, you’ve already used this function. In this chapter, however, you’ll start to see how it is meant to be used.

Trace Output

Just to reiterate, you can trace in haXe, whether it is a Flash, JavaScript, or Neko application, using the trace function. This function is used in the same way for each platform, though the method of its display is different in each. For completeness, this chapter will run through tracing out text for each platform.

Flash

As previously mentioned, the output from a trace function, when used in a Flash application, is written to a TextField object that is overlaid above the entire movie. The field is scrollable, so if the amount of text gets too long, you can scroll through the text to read the part of interest.

Neko

Neko apps write to the console window in the same fashion as the C printf function when used in a Neko ndll module. If used as a web application, the trace output is instead mapped to the output of the web page.

One point worth remembering is that, should you use the Neko framework in a desktop application and restrict the command (shell) window from opening, you will be unable to see any trace output generated by the application.

JavaScript

Outputting a trace value in JavaScript requires a little work. As generated JavaScript code requires a container HTML page in order to function, it is necessary to add a div or other such tag to the HTML markup with the id haxe:trace. This then facilitates as an object that the trace output can render to.

```html
<html>
<head>
    <title>Trace Output</title>
</head>
<body>
    <div id="haxe:trace"></div>
    <script type="text/javascript" src="helloworld.js"></script>
</body>
</html>
```

Later you will see how the trace function can be modified so that the output is routed elsewhere—to an alert box, perhaps.
Chapter 7: When Things Go Wrong

**haXe Trace versus ActionScript Trace**

You ActionScript folk might be itching to skip this section and move on to exception handling, but it is recommended that you spare just a little time reading this chapter thoroughly. There is much more to tracing in haXe than in ActionScript that extends beyond the output panel.

In ActionScript, the `trace` function was really quite simple and limiting. So, you can pass a string to an output window. Big deal. Sure, this is kind of an invaluable feature, but in all of the past eight versions of Flash using the original ActionScript Virtual Machine, it hasn't evolved enough to provide some other basic features that lend themselves well to finding those troublesome bugs without having to result to stepping into the code. Thankfully, however, Nicolas Cannasse has addressed this issue nicely, and has given pretty much all the functionality one could want in a `trace` function. For a start, you're no longer left guessing where your `trace` output is coming from, as each `trace` response is decorated with the associated class name and line number of the `trace` call. This saves loads of time in decorating your `trace` calls yourself so that you can tell one `trace` from another. Another great feature is being able to reroute the `trace` output, so you can stream it to an external file or GUI container. You can even choose to rewrite the functionality of the `trace` function altogether, if you like. Also, unlike the pesky Flash version of the `trace` function, when you pass an object to a haXe `trace`, it will iterate through each property of the object and attempt to convert the property’s value to a string representation; then, if any nested objects are found, it will break these up, also. This makes your output so much more usable than the useless `[Object Object]` `trace`.

**The haxe.Log Class**

The haXe `trace` function is actually a simple wrapper for lower level functions supported by each platform. There is also a second static version of the higher level `trace` function belonging to the `haxe.Log` class that provides a gateway to this functionality. The difference here is that the `haxe.Log.trace` function can accept a number of other parameters, while the standard `trace` function usually only expects one parameter: the string you want to output.

The `haxe.Log` class is the main proxy for the `trace` features in each platform supported by haXe. If you want more control over the output of your `trace` functionality, then the `haxe.Log` class is the way to apply it. Table 7-1 lists the functions provided by the `haxe.Log` class.

**Table 7-1**

<table>
<thead>
<tr>
<th>haxe.Log Method</th>
<th>Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>trace( val : Dynamic, pos : haxe.PosInfos ) : Void</code></td>
<td>Flash, JS, and Neko</td>
</tr>
<tr>
<td><code>clear() : Void</code></td>
<td>Flash, JS, and Neko</td>
</tr>
<tr>
<td><code>setColor( color : Int ) : Void</code></td>
<td>Flash only</td>
</tr>
</tbody>
</table>

**haxe.Log.trace()**

As mentioned previously, this version of the haXe `trace` function provides more control over the output. There are typically two ways that this can be done; one way is to pass the function a secondary argument in the shape of an object derived from the `haxe.PosInfos` typedef, while the other route...
Part I: The Core Language

involves supplying a new function mapping the existing `trace` function signature and facilitating the functionality required by your application.

The signature for the `haxe.Log.trace` function looks like the following:

```haxe
```

The `Dynamic` parameter accepts any type you wish to output in your trace, whether this is a string, integer, or object. For very complex custom objects, it is often a good idea to provide a `toString()` method that will parse the data in a way that makes sense to you. You could then pass the return value of this method to the `trace` function. However, if you omit such a method, the `trace` function will do its best to describe the object in as much detail as it can. If a `toString()` method exists in your object, the `trace` function will automatically utilize it, so you do not have to do this explicitly.

The second parameter of the `trace` function is a typedef containing the following data types:

```haxe
typedef PosInfos = {
    var fileName : String;
    var lineNumber : Int;
    var className : String;
    var methodName : String;
    var customParams : Array<Dynamic>;
}
```

The first four properties of this typedef are pretty self-explanatory. They each represent the filename, line number, class name, and method name, in that order, of the location where the `trace` function is called. By submitting a replacement for this object, it is possible to exchange the values of these properties for your own values, though quite frankly, it is hard to see a valid need beyond custom formatting.

The last available property of the `PosInfos` typedef represents any other parameters you want passed to the function. For example, perhaps you want to pass a flag that notifies your custom `trace` function of where you want the output to be directed.

`here`

When you submit a call to the standard `trace` function, you are omitting the second optional parameter present in the `haxe.Log.trace` function signature. However, the data required by the lower level `trace` function that is normally supplied by this parameter is actually still passed via an identifier called `here`. This identifier is a representation of the `haxe.PosInfos` typedef, and provides information pertaining to the exact point in your code that the identifier is used. You can use the `here` identifier elsewhere in your code, if necessary.

The `here` identifier passed with the standard `trace` can also piggyback the custom parameters available to the `PosInfos` typedef you supply to `haxe.Log.trace`. These custom parameters are supplied to the standard `trace` function as individual parameters rather than items in an array, though they are still accessed from the custom parameters array in any custom `trace` handler.

Now try writing your very own `trace` handler. This example won’t be very useful, but should give some idea as to how you can handle your own `trace` calls.
Chapter 7: When Things Go Wrong

class TraceHandler
{
    public static function main()
    {
        haxe.Log.trace = myTrace;
        trace( "Can I trace it?" );
        var pi : haxe.PosInfos = {
            className : "MyClass",
            fileName : "MyClass.hx",
            methodName : "someFunc",
            lineNumber : 50394,
            customParams : ["let's find out..." ]
        );
        haxe.Log.trace( "Yes I can!", pi );
        trace( "Definitely!!!", "Are you sure?", "Just to be certain..." );
    }
    public static function myTrace( v : Dynamic, ?pos : haxe.PosInfos ) : Void
    {
        var f : flash.MovieClip = flash.Lib.current;
        if ( !Std.is(f.__txtField__, flash.TextField ) )
        {
            f.createTextField( "__txtField__", 9999, 0, 0, flash.Stage.width,
                                flash.Stage.height );
        }
        for ( i in pos.customParams )
        {
            f.__txtField__.text += pos.fileName + "- > " + pos.className + "- > " +
                                  pos.methodName + "- > line(\" + pos.lineNumber + \") : \" + i + \"\n";
        }
        f.__txtField__.text += pos.fileName + "- > " + pos.className + "- > " +
                               pos.methodName + "- > line(\" + pos.lineNumber + \") : \" + v + \"\n";
    }
}

Having compiled and run this code for the Flash platform, you should now find that, when calling the
trace function, you are provided with more data about the location of the trace calls. Of course, a
couple of these lines are bogus, but they do provide you with an overview of the amount of power you
have over how these function calls are treated.

Here is what you should see when you run the preceding code in Flash:

TraceHandler.hx- > TraceHandler- > main- > line(6) : Can I trace it?
MyClass.hx- > MyClass- > someFunc- > line(50394) : let's find out...
MyClass.hx- > MyClass- > someFunc- > line(50394) : Yes I can!
TraceHandler.hx- > TraceHandler- > main- > line(14) : Are you sure?
TraceHandler.hx- > TraceHandler- > main- > line(14) : Just to be certain...
TraceHandler.hx- > TraceHandler- > main- > line(14) : Definitely!!!

The first part of the code to look at is the myTrace function, as it is here that most of the work is carried
out. This function is assigned as the body code for the haxe.Log.trace function, and overwrites the
original functionality. The myTrace function then facilitates the handler for the trace and haxe.Log
.trace function calls.
Part I: The Core Language

When `trace` is now called, the `myTrace` function first checks for the existence of a `TextField` object on the Flash stage with the name `__txtField__`, and creates one if it is not found. Then, it loops through any custom parameters passed with the function call, and displays them inside the `TextField`. Lastly, it displays the primary item as a string in the `TextField`.

When writing data to the `TextField`, the `myTrace` function constructs a string using the data provided by the `haxe.PosInfos` typedef. In the case of the second call to the `trace` function, bogus information was sent in a custom object with the `haxe.PosInfos` signature. This can sometimes prove a valuable trick when specific data is required within the `trace` output that is not otherwise forthcoming.

**haxe.PosInfos**

You’ve now seen how `haxe.PosInfos` can be used alongside the `here` identifier when overriding the available haXe `trace` functionality. However, it is also possible to use `haxe.PosInfos` in your own functions. While `here` presents detailed data of the location where it is used, `haxe.PosInfos` will provide the same data about the location where its containing function was called, providing the `haxe.PosInfos` is used as an unpopulated optional parameter for that function. This is important, as passing `null` for this parameter will provide undefined data.

Here is an example class using `haxe.PosInfos`, albeit not in a very useful way.

```haxe
class LogHandler
{
    public static function main()
    {
        log( "some data" );
    }
    public static function log( data : Dynamic, ?i : haxe.PosInfos ) : Void
    {
        trace( "logging: " + i.methodName + " (" + i.lineNumber + ") : " + data );
    }
}
```

One very handy use for this feature would be if you wanted to provide file-based logging support for a compiled Neko library, but didn’t want to commandeer the developers `trace` support or expose your own code.

**clear() and setColor()**

The `clear` and `setColor` functions are the remaining two static members of the `haxe.Log` class. Both relate to the `trace` output of your applications, though while `clear` is cross-platform, `setColor` will only function for Flash-related output. These functions are quite self-explanatory, and deserve little attention other than to know that they are at your disposal.

The `setColor` function affects the color of the text displayed in the output to a `TextField`. This function is not always usable, however, especially if you opt to create your own `trace` handler. The `clear` function, on the other hand, is used when you wish to purge the `trace` output, and is invaluable when using the `trace` function substantially.
Chapter 7: When Things Go Wrong

To Trace or not to Trace

The `trace` function is pretty flexible, especially when compared to the current offering as supplied by the Flash IDE. However, there is little supplied that couldn’t be packaged up in your own classes. So why use the haXe `trace` function at all? Well, why not? would be one argument. After all, it certainly is usable, and there is no point in reinventing the wheel. However, there is another feature that makes the `trace` function near perfect. The `--no-traces` compiler switch.

Suppose you’ve created a large haXe application that used your own debugging framework extensively, which involved writing data to the screen. Now, you want to output a final version of this application for deployment. The problem is, you must now comment out each line of code where these function calls take place. And, even if you provide a method to omit these messages, your code will still be bloated with all the unused debug code floating uselessly in your release file. The `--no-traces` compiler directive, on the other hand, ensures that all `trace` code is not compiled at all, leaving your release file as slimline as you can possibly make it, without having to alter a single line of code. Now that’s cool!

The `--no-traces` compiler switch will only exclude calls to the standard `trace` function from compilation. All calls directly to the `haxe.Log.trace` function will still be compiled as normal. This is why we consider the haXe `trace` function near perfect.

Exceptions

Exceptions are often misunderstood, and many developers consider exceptions as errors in their code that they can recover from. Now, depending on your capabilities as a programmer, this might very well be the case. However, you know from the first part of this chapter how to find bugs in your code and remove them, so why handle them twice?

Take a look at this from another perspective. What does the word exceptions mean to you? It should mean something that doesn’t fit a given set of rules, as in “the exception to the rule.” If you imagine your code as a series of rules, then an exception is something that doesn’t fit well inside of it. Now, this doesn’t have to be an error, merely an event within your code that a specific block was not designed to handle at the time and place of its inception. Also, this means that, should such an event arise, it should not necessarily be handled as an error, but possibly as an event that should be trapped and handled elsewhere.

Numerous philosophies are oriented toward programming with exceptions. Many will claim that trapping exceptions and rethrowing them to a parent object as a way of handling a specific non-error is a very sloppy way of coding. Many others would consider trapping exceptions as some form of a virtual post office that facilitates an alternative to checking the return values of each and every function call while providing less code that is more pleasing to the eye. Regardless of how you see them, however, exceptions are a necessity in programming, and can prove a very powerful tool when working in an object hierarchy.

Those of you who have dealt with programming in Java or the .NET framework will know exceptions very well. Pretty much everything you do that generates an error will produce an exception. However, it is important to understand that exceptions in these languages are objects created by other objects, often buried deep within the language framework. Once created, they are bubbled up through the application until caught and dealt with, either by other objects in the language framework, or by your own classes. If
Part I: The Core Language

not dealt with properly, the exceptions are eventually exposed in a message output and your application is often brought to a halt.

Languages that don’t use an exception framework will often provide a more subtle way of alerting the developer to an error. Some functions provided by the language may return a null or negative integer value, while other more severe errors may bring the operating system to a complete standstill. If you think about it, though, returning a negative integer from a function is not exactly a descriptive way of alerting a developer to errors. For instance, it may be fine to assume that a negative value denotes an error while a positive number denotes success, but where, then, does zero fit into this? Should you consider it an error, or a successful return value? The fact is, it could mean either, depending on the rules of the functions you’re calling.

Another issue you should consider is, what happens if there are several ways that a function can cause an error? Do you return different negative values depending on that error? Again, this is not a very descriptive option. This is why exceptions are so valuable, and why you should make use of them in your own applications.

Dealing with Exceptions

So now you know why you should use exceptions. Unfortunately, while haXe is a language with exception support, the supported platforms don’t really see eye to eye on how best to implement them. When Neko was written, Nicolas Cannasse decided that any uncaught exception should be exposed to the user and the program that generated the exception should end. This is the belief of many desktop-oriented language authors. However, Flash isn’t so forthcoming and would sooner pretend the exception didn’t occur than allow the exception to cause any untoward issues. A guess for this kind of “blind” style exception handling is that when the Flash virtual machine was developed, Macromedia (or perhaps Allair—the original creators of Flash) did not want Internet surfers to be alarmed by exceptions appearing because of poorly written Flash movies. If you consider the programming capabilities of the original developers of Flash movies during the infancy of the ActionScript language, this was probably a wise choice by Macromedia. However, this kind of decision doesn’t help those developers that would welcome such support in the more recent renditions of ActionScript.

So, now that you can see what you’re up against, take a look at both Neko and Flash when generating an exception:

```haxe
class UncaughtException
{
    public static function main()
    {
        var t : Array<String>;
        t.push("me");
    }
}
```

Compile the preceding class for both Neko and Flash, and then run them both. When run, the Flash player should display a blank screen, while the Neko application will generate the following text:

```
Called from <null> line 1
Called from UncaughtException.hx line 6
Uncaught exception - Invalid field access : push
```

186
Chapter 7: When Things Go Wrong

This is an uncaught exception, which means, it is an exception that you have not caught and dealt with in your code. The problem with the preceding class is that the Array wasn’t instantiated before the method push was called, so, as far as the virtual machine is concerned, there is no method called push available. Now, the fact of the matter is, the Flash virtual machine would have hit the same wall that the Neko virtual machine encountered, except that the Flash virtual machine chose to ignore the error and continue as normal. Does this mean the exception was even generated? Maybe, but, like most invisible pests, you’ll need to catch one to prove it exists.

**Catching an Exception**

Exceptions are a little like buses. You know one will appear at some point, though you wouldn’t like to hazard a guess at when, and when it does arrive, it’ll quickly be followed by several others. The nice thing about this simile, though, is that all you have to do to catch one is to stick out your thumb. Well, okay, not quite, but it’s certainly no harder than that.

So, how are exceptions caught? Ironically, you catch them with the catch keyword, but to catch them, you must first try a block of code using the try keyword. Here’s a look at this using the previous example:

``` ActionScript
class CaughtException {
    public static function main() {
        try {
            var t : Array<String>;  
            t.push("me");
        }
        catch ( err : String ) {
            trace( err );
        }
    }
}
```

As you can see, the only difference between this example and the previous one is that you’ve now surrounded the original content of main with a try block and then proceeded that with a catch block. Now, when run, the virtual machine should attempt to execute the two lines of code, and then if an exception arises, it will be caught with the catch block and dealt with.

Compile this class now for both the Flash and Neko virtual machines. If successful, your Neko output should now present just the one line of text; the specific error found in the main function. However, the Flash player still doesn’t output anything. As discussed earlier, exceptions are part of an object framework oriented at dealing with errors and unexpected events, so it would seem that the Flash player does not deal with these by default.

So, how do you deal with exceptions in a virtual machine that doesn’t seem to throw any? The answer is to throw your own.
Part I: The Core Language

**Throwing Exceptions**

To throw an exception means to create an exception object and launch it into the exception handling framework. Now, this might sound like playing catch with oneself, which it indubitably is, but if that ball returns to you every time with some useful information, then it’s a game worth playing.

In order to throw an exception, you need to know when to throw it and how. Now, understanding how to throw an exception is easy. You simply use the `throw` keyword, followed by the information to throw:

```python
throw "Catch this!";
```

Simple, huh? Now you just need to tackle *when* to throw it. You can’t just throw them willy-nilly, as you’ll end up causing unnecessary panic. However, by not throwing an exception, you may be withholding valuable information that other classes will rely on. So, what do you do?

The answer is often down to personal preference. Usually, the best course of action is first to decide on the urgency of the error at hand, then, if there is absolutely no way for the current function to deal with the issue and the process requested is of an urgent nature, then an exception must be thrown. But, in order to do that, you will need to find out if an error is present, which requires data checking. Take a look at the `CaughtException` class using the `throw` keyword:

```python
class CaughtExceptionB {
    public static function main() {
        try {
            var t : Array<String>;
            if (t != null)
                t.push("me");
            else
                throw "Array t is not an object";
        }
        catch ( err : String )
        {
            trace( err );
        }
    }
}
```

Now, if you compile and run this class in Flash, you should finally see your exception printed to screen. Great, isn’t it? The information presented is limited, but you’ll look into extending this. The main point is that you can now fire a complaint whenever certain criteria is not met, which can then be dealt with, either at the end of the particular method, or by a class further down the object hierarchy.

**Handling Different Exception Types**

So, now you know how to throw and catch exceptions, but what use are they? Until this point, you have been dealing specifically with thrown `String` exceptions, but you can ultimately throw any type of object you like. What’s more, you can specify different `catch` blocks for each type of exception that will be thrown. For instance, within a `try` block, you might have two possible areas where an integer will be
Chapter 7: When Things Go Wrong

thrown and another three where a string will be thrown. Therefore, you can provide two catch blocks: one for the thrown strings and one for the integers:

```haxe
try
{
   // ...
}
catch ( err : String )
{
   trace( "Error: " + err );
}
catch ( line : Int )
{
   trace( "Error found on line " + line );
}
```

If you happen to be unsure of what exception type might be thrown, or you want to provide a block of code to handle all incoming exceptions, regardless of their type, then you can specify Dynamic as the exception type. Dynamic facilitates a catchall block, which means that any exception thrown by the code nested inside the try block will end here. If you still want to catch specific exception types first, then you can place these before the Dynamic catch so that they are dealt with before the exceptions ever reach the Dynamic catch.

**Handling Complex Exceptions**

Handling primitive exceptions is not a very useful feature, unless you simply wish to output a little text when an exception is thrown, but being able to filter what exceptions are caught is very useful indeed when applied to objects. For example, it could be that you develop a set of classes or typedefs oriented to containing information specific to errors. This way, not only can you throw more information on a particular error, but you could also group error types using a class hierarchy and handle them in a logical order.

Here’s a look at this in an example. First, you’ll need a generic error object. This object will need to transport any type of error possible, so should provide a simple but extendable feature set:

```haxe
class GenericError
{
   public var message( getDescription=null ) : String;
   private var __description : String;
   private var __info : haxe.PosInfos;
   public function new( desc : String, ?info : haxe.PosInfos ) : Void
   {
      this.__description = desc;
      this.__info = info;
   }
   private function getDescription() : String
   {
      var msg : String = "Class : " + this.__info.className + " -> ";
      msg += this.__info.methodName + "()\nline ";
      msg += this.__info.lineNumber + " : " + this.__description;
      return msg;
   }
}
```
Part I: The Core Language

Now, you’ll require a class that extends the base error class.

```haxe
import GenericError;
class ObjectNotInstantiated extends GenericError
{
    public function new( ?info : haxe.PosInfos )
    {
        super( "Item is not an instance of an object" );
    }
}
```

That’s all you need for the error classes. How about a test class to run your little experiment?

```haxe
import GenericError;
import ObjectNotInstantiated;
class ErrorObjectTest
{
    public static function main()
    {
        try
        {
            throw new ObjectNotInstantiated();
        }
        catch( err : ObjectNotInstantiated )
        {
            trace( "Not an instance: " + err.message );
        }
        catch( err : GenericError )
        {
            trace( "Generic: " + err.message );
        }
    }
}
```

Now, if you compile these, you should get the following print to screen:

```
ErrorObjectTest.hx:14: Not an instance : Class : ObjectNotInstantiated
-> new()
line 7 : Item is not an instance of an object
```

To get different results, try throwing an instance of `GenericError` instead (remembering to pass a descriptive string). Alternatively, try rearranging the catch statements, so you can see how each error type is filtered.

All you have really done here is create a bare-bones exception class that any new exception class can derive. The `GenericError` class includes a parameter of type `haxe.PosInfos` in the constructor, so that, when instantiated, the exception will be able to provide a detailed account of where the exception was thrown.

Now, because any class extending the `GenericError` class will essentially be a new type all of its own, you can specifically target these classes inside a catch statement. Alternatively, specifying the `GenericError` class itself will target this and all derivatives of this class, providing a complete error-trapping framework.
Chapter 7: When Things Go Wrong

Catching Unexpected Exceptions

As a fair and well-organized programmer, you have to be sure that all exceptions thrown from your code will be fully documented so that any other developer utilizing your code will have prepared catch statements for every eventuality. However, there are times when an exception slips through the net causing serious problems for the user of the application, and even more serious implications for the developer who has just received a nasty letter describing how a user had recently lost hours of work because of some unexplained software bug, and to expect a bill for loss of earnings. Not nice.

This sort of situation could happen to anyone (or at least those of you who haven’t thought through your software licensing), and indeed, you could argue that, had the developer used a catch statement for the Dynamic type, thus catching all further exceptions for a try block, then this bug might not have happened. But what if the exception was thrown from someone else’s code that you yourself utilized and was not surrounded within a try block at any point while it bubbled its way through your class hierarchy?

Let’s pause for a moment and consider this problem. The main purpose for catching exceptions is to provide code for certain eventualities that the main body of code was not designed to handle. This could be handled in a parent class, or within the method of the exceptions inception. However, all exceptions should at least be expected to some extent. For example, while dealing with files, you should expect the occasional file not found or permission denied errors. Thus, by providing catch statements, you are able to deal with these eventualities. However, should a completely unexpected error occur in an unexpected place, you should at least be able to save what work you can, clean up as many objects as possible, and prepare for the worst. Luckily, haXe provides a means to do just that, albeit only for Flash and JavaScript, using flash.Lib.setErrorHandler and js.Lib.setErrorHandler.

```actionscript
static function globalHandler( msg : String, stack : Array<String> )
{
    trace( "Error : " + msg );
}
static function main()
{
    flash.Lib.setErrorHandler( globalHandler );
    throw "Some error";
}
```

The purpose of these functions is to provide a means to set a method within your application that can specifically handle any uncaught exceptions. These handlers trap only exceptions that bubble up through the parent class of the handler, so any exceptions requiring capture on a global scale will need to be defined within the main application class.

The CallStack and ExceptionStack

Receiving information about an error is all well and good, but very often you need a little more information to help resolve those troublesome bugs. For instance, when you catch an exception, you are able to discover what you know about the event via the custom exception message. You can also find out
Part I: The Core Language

where and when the exception was thrown. What you can’t necessarily find out, however, is what led to
the exception in the first place.

To help with this issue, haXe provides two cross-platform functions, haxe.Stack.callStack() and
haxe.Stack.exceptionStack(), that are available when the -debug compiler switch is enabled.
These clever little functions provide critical information where and when you need it, though they’re
probably not the kind of functions you’d like to leave active in a release version of your applications.

**haxe.Stack.callStack()**

This function is best used when you absolutely must see what is happening inside your application, but
at a time when an error—in the classic meaning of the word—is not evident. For example, you may be
trying to trace the journey of a variable value so that you can work out why a particular calculation is
not behaving quite as it should.

While not able to return the value of the variable on its journey, this function can at least list each of the
functions called from the point of entry in the base class to the moment that the callStack() function is
called. Once the journey from the base class returns, the stack is reset for the next called function.

The signature for the callStack() function is:

```
    haxe.Stack.callStack() : Array<haxe.StackItem>;
```

As you can see, when called, an array listing the call stack details using the haxe.StackItem
enumerator is returned. The haxe.StackItem enum, at least in this case, will return the class and
method names of the call. However, if you would rather just dump the contents of each StackItem, you
can do so using the haxe.Stack.toString() method.

Let’s see this in an example:

```haxe
class CallStack
{
    public static function main()
    {
        var i : Int = 0;
        i = add( i, 6 );
    }
    public static function add( a, b )
    {
        return a + minus( b, 2 );
    }
    public static function minus( a, b )
    {
        return a - multiply( b, 2 );
    }
    public static function multiply( a, b )
    {
        return a * traceStack( b );
    }
    public static function traceStack( a )
```

```haxe
class CallStack
{
    public static function main()
    {
        var i : Int = 0;
        i = add( i, 6 );
    }
    public static function add( a, b )
    {
        return a + minus( b, 2 );
    }
    public static function minus( a, b )
    {
        return a - multiply( b, 2 );
    }
    public static function multiply( a, b )
    {
        return a * traceStack( b );
    }
    public static function traceStack( a )
Chapter 7: When Things Go Wrong

```haxe
{ var cs = haxe.Stack.callStack();
  trace( haxe.Stack.toString( cs ) );
  return a;
}
```

If you compile this class and run it, you should be presented with the following output:

```
CallStack.hx:27: Called from CallStack method traceStack
Called from CallStack method multiply
Called from CallStack method minus
Called from CallStack method add
Called from CallStack method main
```

Reading backward through the stack, you can see that you started at the method `main` in the `CallStack` class. From there, the `add` method was called. That method then called the `minus` method, which in turn called the `multiply` method, which finally called the `traceStack()` method. Although you could easily have recreated this scenario with a `trace` call in each method, the `callStack()` function has enabled you to print this information cleanly, without having to write extra code in multiple locations. What’s more, you can guarantee that no method in the call stack is left out.

**haxe.Stack.exceptionStack()**

The `exceptionStack` function works in a very similar fashion to the `callStack()` function. However, while the `callStack()` function is intended for use as a way to track which methods are called from the base class to the moment the `callStack()` function is called, the `exceptionStack()` function is used to trace the journey from the method where an exception is thrown to the method where the `exceptionStack()` function is called:

```haxe
class ExceptionStack
{
  public static function main()
  {
    try
    {
      var i : Int = 0;
      i = add( i, 6 );
    }
    catch ( err : Dynamic )
    {
      var es = haxe.Stack.exceptionStack();
      trace( haxe.Stack.toString( es ) );
    }
  }
  public static function add( a, b )
  {
    return a + minus( b, 2 );
  }
  public static function minus( a, b )
  {
    return a - multiply( b, 2 );
  }
}
```
public static function multiply( a, b )
{
    throw "Darn, we have an error";
    return a * b;
}
}

When compiled and run, you should be presented with the following:

ExceptionStack.hx:13: Called from ExceptionStack method multiply
Called from ExceptionStack method minus
Called from ExceptionStack method add
Called from ExceptionStack method main

If you compare this output to that of the callStack() function, you should notice that the journey is a reverse image, although without the traceStack() method call.

The Exception Master Class

Throughout this chapter, you looked at some very powerful tools to aid in combating errors in your code, whether they’re structural errors you could have dealt with, or system errors that you as a developer have no control over. While only experience and hard work can help you deal with the former issue, the haXe exception features are your only real safeguard against the latter.

If you take a look at the exception handling capabilities of languages such as Java and C#, you’ll notice that they share a similar trait. In both of these languages, and others like them, exception classes are used and served from the very most inner classes in the language framework. As exceptions are designed to bubble up through the hierarchical structure until handled, therein lies their power to provide detailed information that can aid you in providing adaptive functionality rather than inferior software. However, in order to facilitate the usefulness of exceptions within your code, it would help to extend from a decent functional base exception class, similar to the GenericError class attempted earlier in this chapter. You’ll now undertake a second attempt to produce a more complete base exception class for use in future examples in this book, utilizing most of the features you have learned in this chapter.

Creating a Complete Exception Class

The more you use the haXe language, the more likely you will continue to modify and extend an array of classes that you have created yourself and that do not exist in the main haXe library. As error and exception handling are often highly individual to the developer, it makes sense for your custom class arsenal to contain an all powerful exception class for use in your everyday applications. As time goes on, you may find it necessary to add platform-specific code that can deal with tasks such as saving log information to a text file, posting exception data from a Flash movie back to the server for cataloging, and even developing a third-party trace or object viewer application. All is certainly possible with haXe. However, for now, it is important to get the fundamental features required to make your development life easier.
Chapter 7: When Things Go Wrong

The Bare Necessities

As a starting point, take the base code from the GenericError class you constructed earlier. Granted, this class is pretty basic, but it does facilitate some of the primary required functionality for your master class.

It is a good practice to name this class Exception, as it is a base class for all future exception objects. Also, note that the return value for the message getter function has been drastically reduced. This is so you can provide numerous getters for each information type, so as to provide a more custom style functionality. Many developers find that, under normal circumstances, a simple message is sufficient while providing alternative code for intercepted exception objects. However, extensive information is still required for tasks such as logging and simple bug finding.

```haxe
class Exception
{
    private var __description : String;
    private var __infos : haxe.PosInfos;
    public var message(getMessage,null) : String;
    public function new( msg : String, ?info : haxe.PosInfos )
    {
        this.__description = msg;
        this.__infos = info;
    }
    public function getMessage() : String
    {
        return __description;
    }
}
```

As you can see, the message getter now merely returns the message string passed into the constructor of the class. In order to provide actual information collected by the virtual machine, you need to add a few more functions.

```haxe
public function getSource() : String
{
    var src : String = "File: " + __infos.fileName + " | Line: " + __infos.lineNumber + "\n";
    src += "In method " + __infos.methodName + " of class " + __infos.className;
    return src;
}
```

This function caters for the second of the two constructor parameters, the haxe.PosInfos object, as normally provided by the virtual machine itself. This should now cover 90 percent of the exception class needs. However, you should also include stack trace information when available.

Debug Level Functionality

You probably won’t want to release software compiled with the -debug flag. For one, it exposes information about your application that you otherwise wouldn’t want exposed, while at the same time, it bloats your application slightly with extra code. However, none of this will matter when debugging your application, so any added functionality, however small, could be code well implemented.
The only real benefit to using the -debug flag with regard to exception handling is that of the callStack and exceptionStack functions. Supposedly, the flash.Lib.setErrorHandler() function also requires the -debug flag set, though you have not seen any benefits in doing so.

```haxe
public function getStack() : String
{
    var es = haxe.Stack.exceptionStack();
    var str : String = StringTools.rpad( "StackTrace\n", "=", 21 ) + "\n";
    str += haxe.Stack.toString( es );
    return str;
}
public function getCallStack() : String
{
    var cs = haxe.Stack.callStack();
    var str : String = StringTools.rpad( "CallStackTrace\n", "=", 26 ) + "\n";
    str += haxe.Stack.toString( cs );
    return str;
}
```

These two functions will provide function call and exception stack information when required, but will return undefined when the -debug flag is not provided. As -debug is a compiler flag, however, you can make use of compiler directives and choose to return a polite string should the exceptionStack() and callStack() functions be unavailable. Here are the same two functions using a conditional statement compiler directive:

```haxe
public function getStack() : String
{
    var str : String = "Exception stack available in debug mode only.";
    #if debug
    var es = haxe.Stack.exceptionStack();
    str = StringTools.rpad( "StackTrace\n", "=", 21 ) + "\n";
    str += haxe.Stack.toString( es );
    #end
    return str;
}
public function getCallStack() : String
{
    var str : String = "Call stack available in debug mode only.";
    #if debug
    var cs = haxe.Stack.callStack();
    str = StringTools.rpad( "CallStackTrace\n", "=", 26 ) + "\n";
    str += haxe.Stack.toString( cs );
    #end
    return str;
}
```

As you can see, if the -debug flag is not set, then a default string value will be returned from the function call, instead.

Now, this is all well and good, but there is a slight issue, here. It could very well be some time between when the exception is created and thrown and when it is caught. This means that, while the
exceptionStack() will now prove its usefulness, the callStack() function suddenly becomes pointless, as any function listings it does provide will include calls that exist after the exception was thrown, not the list of functions executed on the run up to the exception objects instantiation. Don’t panic, though, as this is easily rectified. You simply provide the functionality of the getCallStack() method to the class constructor. This way, the list of functions provided by the callStack() function can be stored in a string member variable and returned in the getCallStack() method call:

```haxe
public function new( msg : String, ?info : haxe.PosInfos )
{
    __description = msg;
    // string member variable to store callStack data
    __calls = "Call stack available in debug mode only.";
    #if debug
        var cs = haxe.Stack.callStack();
        __calls = haxe.Stack.toString( cs );
    #end
    __infos = info;
}
public function getCallStack() : String
{
    return StringTools.rpad( "CallStackTrace
", "=", 26 ) + "\n" + __calls;
}
```

### Providing a Generic Output Function

Every good object class deserves a toString() method. This way, should the developer wish to simply dump every bit of information possible in an easy to read format, he or she doesn’t have to parse the information. Thankfully, this should be easy to provide for this class, as every other member method outputs a string value. All you have to do is to concatenate the output of each function into a single return value, like this:

```haxe
public function toString() : String
{
    var str : String = getMessage() + "\n" + getSource();
    str += "\n\n" + getStack();
    str += "\n\n" + getCallStack();
    return str;
}
```

You shouldn’t need to worry about whether the -debug flag is set for the getStack() method call, as the function should already handle this for you.

### The Exception Class Code

That completes the Exception class. Later in the book, you revisit this class and add a few extra features, but for now, here is the class in its entirety:

```haxe
class Exception
{
    private var __description : String;
    private var __infos : haxe.PosInfos;
    private var __calls : String;
}
```
(continued)

public var message(getMessage, null) : String;
public var source(getSource, null) : String;
public var stackTrace(getStack, null) : String;
public var callStack(getCallStack, null) : String;
public function new(msg : String, ?info : haxe.PosInfos)
{
    __description = msg;
    __calls = "Call stack available in debug mode only."
#if debug
    var cs = haxe.Stack.callStack();
    __calls = haxe.Stack.toString(cs);
#if debug
    __infos = info;
#endif
    public function getMessage() : String
    {
        return __description;
    }
    public function getSource() : String
    {
        var src : String = "File: " + __infos.fileName + " | Line: " +
            __infos.lineNumber + "\n";
        src += "In method " + __infos.methodName + " of class " +
            __infos.className;
        return src;
    }
    public function getStack() : String
    {
        var str : String = "Exception stack available in debug mode only."
#if debug
    var es = haxe.Stack.exceptionStack();
    str = StringTools.rpad("StackTrace\n", "=" , 21 ) + "\n";
    str += haxe.Stack.toString(es);
#else end
    return str;
#endif
    public function getCallStack() : String
    {
        return StringTools.rpad("CallStackTrace\n", "=" , 26 ) + "\n" + __calls;
    }
    public function toString() : String
    {
        var str : String = getMessage() + "\n" + getSource();
        str += "\n" + getStack();
        str += "\n" + getCallStack();
        return str;
    }
}

You can test this class by compiling the following code into an SWF file. Try recompiling with and
without the -debug flag set, so you can see how the Exception class might function in a live project.
Chapter 7: When Things Go Wrong

```haxe
import Exception;
class ExceptionTest {
    public static function main() {
        try {
            throw new Exception( "Some strange unknown error occurred" );
        } catch( err : Exception ) {
            trace( err.toString() );
        }
    }
}
```

When run successfully, you should be presented with the following output:

```
ExceptionTest.hx:13: Some strange unknown error occurred
File: ExceptionTest.hx | Line: 9
In method main of class ExceptionTest
StackTrace
==========
Called from ExceptionTest method main
CallStackTrace
==========
Called from Exception method new
Called from ExceptionTest method main
```

**Summary**

Although relatively short, you covered some very important topics in this chapter, including:

- How to trace in each platform
- How to modify the trace handler
- How to use `here` and `haxe.PosInfos` for code tracking
- How to use and handle exceptions
- How to create your own exception framework
- How to use the `callStack()` and `exceptionStack()` functions

The next chapter is the start of Part II of the book, where you learn how to use haXe and Neko to build powerful web applications.
Part II

Server Side, JavaScript, and Flash; Oh My!

Chapter 8: Cross Platform Tools
Chapter 9: Building Websites with haXe
Chapter 10: Seperating Design Using Templates
Chapter 11: Performing Server Side Trickery
Chapter 12: Building Interactive Content with Flash
Chapter 13: Replacing the Need for an IDE
Chapter 14: More Interactive Content with JavaScript
Chapter 15: Putting It All Together with haXe Remoting
Cross Platform Tools

Many classes in haXe standard library can be used cross-platform; some of them have already been discussed in the previous chapters and some more advanced ones are explained in the rest of the book. In this chapter you will find a description of some very commonly used classes that must be part of every developer toolkit and in particular they are:

- XML
- Regular expressions
- Timer triggers
- MD5 cryptography hashes

XML

The Extensible Markup Language (XML) is a very well-known syntax used to exchange data in a human readable form. In this section only a brief description of the language is provided because many resources can be found online.

XML Anatomy

XML documents are plain-text files that must be well-formed and may be valid. To be well-formed, an XML document must respect some syntax rules you will find later in this section. A document is also considered valid if it passes a validation test against a validation document, usually written in the Document Type Definition (DTD) format or XML Schema format.

An XML document is composed by a hierarchy tree of nodes; each node has its own syntax that must be strictly respected for the document being accepted by a processor. A processor can be any kind of software that makes use of the data stored in the XML document. Unlike the HTML processors that are capable of ingesting a lot of syntax errors without breaking the page rendering, the XML processor usually refuses to operate on the document if this presents any syntax inconsistencies.
Part II: Server Side, JavaScript, and Flash; Oh My!

The following XML document is a sample XHTML page. It is good to analyze this document because it presents every type of node allowed and highlights the most common traps for users that are new to XML processing.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>Sample XHTML</title>
<script type="text/javascript" src="external.js"> <!-- inline -->
<script type="text/javascript"></script>
<![CDATA[
 // inline
]]></script>
</head>
<body>
<!-- comment here -->
<p>Link to <a href="http://example.com/">example</a>.</p>
</body>
</html>
```

When a processor parses the text above, a hierarchical tree is built. The first root node is a node of type Document. The document has no other properties or characteristics apart from having child nodes.

Two main sections, the prolog section and the element section, compose an XML document. The first contains meta-information about the document and is composed of two optional nodes, the XML Declaration node, and the Document Type Declaration node. In haXe these nodes are known as Prolog and DocType. They can be omitted or just one of the two can be used. If they exist, they must be the first nodes in the document and respect the given sequence. They are not allowed to have child nodes. A DocType can define many rules but they are not considered nodes because they must be parsed by a specific processor.

The Prolog node defines the XML version, usually 1.0, the character encoding, and optionally if the document may exist as standalone, default value is yes. The node content is wrapped in the `<?xml ... ?>` declaration.

```xml
<?xml version="1.0" encoding="UTF-8"?>
```

The DocType node defines the rules for the validation check of the document. The DTD rules can be expressed inline or referenced in an external document as in the example.

The content of a DocType node is wrapped in the `<!DOCTYPE ... >` declaration.

```xml
<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

What is usually underestimated in XML parsing is that whitespaces, and more generally the characters between tags, are nodes on their own. Between the Prolog node and the DocType node there is a node
Chapter 8: Cross Platform Tools

of type parsed character data (PCData) whose value is a newline character. The processor always parses a PCData node.

The element section of an XML document is composed by one mandatory root node of type Element. An element always has a name, html is the one of the root node, and may have zero or more attributes and zero or more child nodes. Elements and tags are covered in more detail in Chapter 9 in the section dedicated to the HTML syntax.

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
...
</html>
```

Another type of node is the Character Data (CData). Unlike the PCData, the content of a CData node is not parsed by the processor and its value is set as in the value of the node. A CData value is contained between the delimiters `<![CDATA[` and `]]>`.

```
<![CDATA[
 // inline
]]>
```

The last type of node is Comment. The processor ignores any comment content. Comments can be located anywhere in an XML document and their value is delimited by `<!--` and `-->`.

In Figure 8-1 you can find the XML document processed by the haXe XML parser and displayed as sequence of nodes. Each line represents a node; they have been indented to show the hierarchy nesting. The whitespaces in node values have been replaced by the \s symbol and newlines by the

Chapter 8: Cross Platform Tools

of type parsed character data (PCData) whose value is a newline character. The processor always parses a PCData node.

The element section of an XML document is composed by one mandatory root node of type Element. An element always has a name, html is the one of the root node, and may have zero or more attributes and zero or more child nodes. Elements and tags are covered in more detail in Chapter 9 in the section dedicated to the HTML syntax.

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
...
</html>
```

Another type of node is the Character Data (CData). Unlike the PCData, the content of a CData node is not parsed by the processor and its value is set as in the value of the node. A CData value is contained between the delimiters `<![CDATA[` and `]]>`.

```
<![CDATA[
 // inline
]]>
```

The last type of node is Comment. The processor ignores any comment content. Comments can be located anywhere in an XML document and their value is delimited by `<!--` and `-->`.

In Figure 8-1 you can find the XML document processed by the haXe XML parser and displayed as sequence of nodes. Each line represents a node; they have been indented to show the hierarchy nesting. The whitespaces in node values have been replaced by the \s symbol and newlines by the

Chapter 8: Cross Platform Tools

of type parsed character data (PCData) whose value is a newline character. The processor always parses a PCData node.

The element section of an XML document is composed by one mandatory root node of type Element. An element always has a name, html is the one of the root node, and may have zero or more attributes and zero or more child nodes. Elements and tags are covered in more detail in Chapter 9 in the section dedicated to the HTML syntax.

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
...
</html>
```

Another type of node is the Character Data (CData). Unlike the PCData, the content of a CData node is not parsed by the processor and its value is set as in the value of the node. A CData value is contained between the delimiters `<![CDATA[` and `]]>`.

```
<![CDATA[
 // inline
]]>
```

The last type of node is Comment. The processor ignores any comment content. Comments can be located anywhere in an XML document and their value is delimited by `<!--` and `-->`.

In Figure 8-1 you can find the XML document processed by the haXe XML parser and displayed as sequence of nodes. Each line represents a node; they have been indented to show the hierarchy nesting. The whitespaces in node values have been replaced by the \s symbol and newlines by the
Symbol. The DocType declaration has been cut in the middle because of its length. Element names are in boldface and their attributes displayed inline in a space-separated list of key-value pairs.

The Flash implementation, based on the Flash XML API simply ignores the Prolog and DocType nodes and how the first script element is treated as an empty node even if it is not. The Flash implementation should contain an empty PCData node. CData nodes are also ignored.

**haXe XML API**

The haXe XML API implementation is based on the class Xml located in the default package of the standard library. The public functions and variables of the class are described in the Tables 8-1 through 8-3. The class also contains a number of static variables describing the existing node types; those variables are usually used to detect the type of node by comparison and are CData, Comment, DocType, Document, Element, PCData, and Prolog.

**Table 8-1**

<table>
<thead>
<tr>
<th>XML Class Static Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static createCData(data:String):Xml</code></td>
<td>Creates node of type CData.</td>
</tr>
<tr>
<td><code>static createComment(data:String):Xml</code></td>
<td>Creates node of type Comment. Not implemented on Flash 6 to 8.</td>
</tr>
<tr>
<td><code>static createDocType(data:String):Xml</code></td>
<td>Creates node of type DocType. Not implemented on Flash 6 to 8.</td>
</tr>
<tr>
<td><code>static createDocument():Xml</code></td>
<td>Creates node of type Document.</td>
</tr>
<tr>
<td><code>static createElement(name:String):Xml</code></td>
<td>Creates node of type Element.</td>
</tr>
<tr>
<td><code>static createPCData(data:String):Xml</code></td>
<td>Creates node of type PCData.</td>
</tr>
<tr>
<td><code>static createProlog(data:String):Xml</code></td>
<td>Creates node of type Prolog. Not implemented on Flash 6 to 8.</td>
</tr>
<tr>
<td><code>static parse(s:String):Xml</code></td>
<td>Creates an XML class instance starting from the text definition passed as argument. If the argument is not a well-formed document, an exception is thrown.</td>
</tr>
</tbody>
</table>

The Xml.parse() method is to generate an Xml structure from an XML document. The haXe processor checks if the document is well-formed; the following document is processed correctly:

```javascript
var xml = Xml.parse("<root></root>");
```

While the following raises an exception on every platform:

```javascript
var xml = Xml.parse("<root></rotten>");
```

Once an XML document is read from an existing source or created new, other nodes can be created using the corresponding createX() method and then appended to a specific node of the document, the function addChild().
### Table 8-2

<table>
<thead>
<tr>
<th>Xml Class Instance Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addChild(x:Xml) : Void</td>
<td>Adds a child node to the current one. This function only works if the current node is of type <code>Element</code> or <code>Document</code> and if the appended node is of the correct type.</td>
</tr>
<tr>
<td>attributes() : Iterator&lt;String&gt;</td>
<td>Returns an <code>Iterator</code> that contains the names of the attributes associated to the current node. It only works if the current node is of type <code>Element</code>.</td>
</tr>
<tr>
<td>elements() : Iterator&lt;Xml&gt;</td>
<td>Returns an <code>Iterator</code> that contains all the nodes of type <code>Element</code> contained in the current node. It only works if the current node is of type <code>Element</code> or <code>Document</code> and returns only the immediate child nodes.</td>
</tr>
<tr>
<td>elementsNamed(name: String) : Iterator&lt;Xml&gt;</td>
<td>Returns an <code>Iterator</code> that contains all the nodes of type <code>Element</code> contained in the current node that have the name equal to the passed argument. It only works if the current node is of type <code>Element</code> or <code>Document</code> and returns only the immediate child nodes.</td>
</tr>
<tr>
<td>exists(att:String) : Bool</td>
<td>Returns <code>true</code> if the current node has an attribute with the name as the passed argument. It only works if the current node is of type <code>Element</code>.</td>
</tr>
<tr>
<td>firstChild() : Xml</td>
<td>Returns the first child node of the current one. It only works when the current node is of type <code>Element</code> or <code>Document</code>.</td>
</tr>
<tr>
<td>firstElement() : Xml</td>
<td>Returns the first child node of type <code>Element</code> or <code>null</code> if no elements exist in the current node. It only works when the current node is of type <code>Element</code> or <code>Document</code>.</td>
</tr>
<tr>
<td>get(att:String) : String</td>
<td>Returns the value of the attribute with the same name as the passed argument; if a matching attribute does not exist, <code>null</code> is returned. It only works when the current node is of type <code>Element</code>.</td>
</tr>
<tr>
<td>insertChild(x:Xml, pos:Int) : Void</td>
<td>Adds a new child node at the specified position. This function only works if the current node is of type <code>Element</code> or <code>Document</code> and if the appended node is of the correct type.</td>
</tr>
<tr>
<td>iterator() : Iterator&lt;Xml&gt;</td>
<td>Returns an <code>Iterator</code> with all the child nodes of the current node. It only works if the current node is of type <code>Element</code> or <code>Document</code>.</td>
</tr>
<tr>
<td>remove(att:String) : Void</td>
<td>Removes the attribute with the passed name from the current node. If the attribute does not exist, nothing happens. This function only works if the current node is of type <code>Element</code>.</td>
</tr>
<tr>
<td>removeChild(x:Xml) : Bool</td>
<td>Removes a child node from the current node. If the child node is not contained in the current node, nothing happens. This function only works if the current node is of type <code>Element</code>.</td>
</tr>
<tr>
<td>set(att:String, value:String) : Void</td>
<td>Sets the value of an attribute; the first argument is the attribute name and the second is the attribute value.</td>
</tr>
<tr>
<td>toString() : String</td>
<td>Transforms the current node in a XML string representation.</td>
</tr>
</tbody>
</table>
A common mistake is to iterate over the nodes using the `iterator()` function expecting a sequence of nodes only of the `Element` type; the `iterator()` function, as the `firstChild()`, returns the full list of all the child nodes that include nodes of type `Prolog`, `DocType`, `PCData`, `CData`, and `Comment`. To retrieve only the node of `Element` type use `elements()` and `firstElement()` respectively.

### Table 8-3

<table>
<thead>
<tr>
<th>Xml Class Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nodeName(getNodeName, setNodeName):String</code></td>
<td>Is the <code>Element</code> name of the node. It only works if the current node is of type <code>Element</code>.</td>
</tr>
<tr>
<td><code>nodeType(default,null) : XmlType</code></td>
<td>Is the node type.</td>
</tr>
<tr>
<td><code>nodeValue(getNodeValue, setNodeValue) : String;</code></td>
<td>Is the node value. It only works if the current node is of type <code>CData</code>, <code>Comment</code>, <code>DocType</code>, <code>PCData</code> and <code>Prolog</code>.</td>
</tr>
<tr>
<td><code>parent(getParent,null) : Xml</code></td>
<td>Is the parent node or <code>null</code> if it is a newly created node not yet appended to any existing structure, or it is the node of type <code>Document</code> (only one node of this kind can exist in the same hierarchy).</td>
</tr>
</tbody>
</table>

All variables in the XML class use method accessors (getters and setters) to handle their values. This technique has been adopted to overcome the differences between the native XML API implementations.

### Traversing a document

Traversing an XML document is easy if you know the structure of the XML you are using. For example, suppose that you have to retrieve the URL of the link in the XML shown in the “XML Anatomy” section. First, the document content must be accessible in the code; usually a good idea is to read it from some external source like a local file or a remote resource but this is quite often platform specific. To have a unified way to access the document, it is embedded in the code using a haXe resource. The XML file is located in an assets folder and embedded at compile time adding the following switch to the compiler command:

```
-resource assets/sample.xml@sample
```

The code to access the link attribute uses many of the fields described previously.

```javascript
class Main
{
    public static function main()
    {
        var xml = Std.resource("sample");
        var doc = Xml.parse(xml);
        var html = doc.firstElement();
        var htmlchildren = html.elements();
        htmlchildren.next(); // skip head element
        var body = htmlchildren.next();
        // the first element child of "body" is "p"
    }
}
```
Chapter 8: Cross Platform Tools

```haxe
// the first element child of 'p' is the link 'a'
var a = body.firstElement().firstElement();
trace(a.get("href"));
}
}
```

The XML traversing can be very verbose and not very elegant when the nodes to navigate are many, an alternative solution is to use the `haxe.xml.Fast` class that is described in Chapter 16 where some more advanced XML techniques are described.

Accessing the text content of an element is simple but you cannot forget that the text is a node itself and must be accessed as such. In the preceding example, to get the inner content of the variable `a`, you cannot use the `nodeValue` variable because this field is not available on nodes of type `Element` and you must use the `firstChild()` function.

```haxe
trace(a.firstChild().nodeValue);
```

If you do not know the exact content of an `Element` node and you want to obtain a string representation of this content, you can write a simple function that works in a similar way to the `innerHTML` property in the HTML Dom (see Chapter 9 for further details).

```haxe
class XmlUtil
{
    public static function innerXML(x : Xml)
    {
        return if(x == null)"
        else if(x.nodeType == Xml.Document || x.nodeType == Xml.Element)
        {
            var b = new StringBuf();
            for(n in x)
                b.add(n.toString());
            b.toString();
        } else
            x.toString();
    }
}
```

**Creating a document**

Creating an XML document presents no difficulties. You only have to ensure that after a node has been created it is added to some node of your document.

```haxe
class Main
{
    public static function main()
    {
        var doc = Xml.createDocument();
        #if (flash9 || js || neko)
            doc.appendChild(Xml.createProlog("<?xml version="1.0"?>"));
    }
}
```
Regular Expression

A regular expression is a text pattern used to describe or match a set of strings. The EReg class provides a cross-platform implementation of this feature. Regular expressions are not supported on Flash versions older than 9. On the supported platforms, the EReg class acts as a wrapper to the native implementations (Neko uses the PCRE library). A regular expression is performed to find a correspondence inside a string; if the correspondence exists, it is said that a match has been found. Informally the regular expression is also known with its abbreviation regexp that will be used for brevity in the rest of the chapter.

The EReg Class

The EReg class in the default package is there where the regex functionalities are implemented. The class methods are described in Table 8-4, and you will find many examples of usages in the rest of the section and in the section dedicated to the regex patterns.

Because regular expressions are so important in everyday programming, haXe has also a specific syntax to declare them that elegantly replaces the EReg constructor.

The regex special syntax is easier to understand with an example; two following lines are equivalent.

```haXe
var re = new EReg("pattern", "gim");
var re = ~/pattern/gim;
```

Note that you can use the canonical dot-syntax also with the regex special syntax.

```haXe
var name = "John Doe";
// check that the var contains the word "John" (case-insensitive)
if(~/<john/i.match(name))
{
    // ... do something here
}
```
Some optional modifiers, described in Table 8-5, can affect the behavior of the pattern. The options must be concatenated in a single string value; the order does not matter.

Before going into details over the definition of patterns, have a look at the following examples to see how and when the EReg methods and options must be applied.

The matched() method returns the content of a group defined in the expression. In the pattern b(c)d the group is between the parentheses; the pattern matches if the entire string bcd is present in the passed string and if this happens, then a group that also contains the c letter exists. Note that the zero
Table 8-5

<table>
<thead>
<tr>
<th>Option</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>global</td>
<td>When used with replace() or split() the operation is repeated for each occurrence of the pattern and will not stop on the first.</td>
</tr>
<tr>
<td>i</td>
<td>case insensitive</td>
<td>The matches are case insensitive.</td>
</tr>
<tr>
<td>m</td>
<td>multiline</td>
<td>The start (^) and end ($) anchors (see below) work on the beginning and end of the string and not on each new line as by default.</td>
</tr>
<tr>
<td>s</td>
<td>single-line</td>
<td>The dot (.) character matches also the newline characters considering the whole string as a single line. This option is only available in Neko.</td>
</tr>
<tr>
<td>u</td>
<td>utf-8</td>
<td>Activates the UTF-8 mode. This only affects the Neko platform because in Flash it is always active and in JavaScript it depends on current page encoding.</td>
</tr>
</tbody>
</table>

Although you can define more than one group for pattern, it is not possible to retrieve more than one occurrence of each group at once. To do so you have to repeat the match() method on the remaining portion of the string using the result of matchedRight() as the test value as illustrated in the following code. The expression looks for any word of length between four and five characters.

```actionscript
class Main
{
    public static function main()
    {
        var re = ~/b(c)d/;
        re.match('abcde');
        trace("string: " + re.matched(0)); // bcd
        trace("group: " + re.matched(1)); // c
        trace("left: " + re.matchedLeft()); // a
        trace("right: " + re.matchedRight()); // e
        var p = re.matchedPos();
        trace("pos: " + p.pos + ", len: " + p.len); // pos: 1, len: 3
    }
}
```

Although you can define more than one group for pattern, it is not possible to retrieve more than one occurrence of each group at once. To do so you have to repeat the match() method on the remaining portion of the string using the result of matchedRight() as the test value as illustrated in the following code. The expression looks for any word of length between four and five characters.

```actionscript
class Main
{
    public static function main()
    {
        var re = ~/\b\w{4,5}\b/;
        var s = "The quick brown fox jumps over the lazy dog";
        var results = new List();
        ```
while (re.match(s))
{
    results.add(re.matched(1));
    s = re.matchedRight();
}
trace(results);  // {quick, brown, jumps, over, lazy }

The split() method uses the pattern expression to divide a string into chunks. The `g` option allows splitting the whole expression and not just the first occurrence.

class Main
{
    public static function main()
    {
        var re = ~/x/g;
        var a = re.split('axbxc');
        trace(a);  // [a, b, c]
    }
}

Groups are not used to split strings; only the whole pattern can be used for this purpose. The following expression defines a group that is not used by the split() method.

class Main
{
    public static function main()
    {
        var re = ~/b(c)d/g;
        var a = re.split('abcdeabcde');
        trace(a);  // [a, ea, e]
    }
}

The replace() method substitutes the matched expression with a second argument passed to the function. The `g` modifier applies the replacement on the whole string.

class Main
{
    public static function main()
    {
        var re = ~/b(c)d/g;
        var s = re.replace('abcdeabcde', 'x');
        trace(s);  // axeaxe
    }
}
Groups can be used in replacement to create back-references. A back-reference contains the value of a matched group and can be used in the replacement string. The back-references have the $ prefix followed by the numeric id of the group. Back-references are handy in replacements when you do not want to replace a portion of the string but want to wrap its value between some characters.

```javascript
class Main {
  public static function main() {
    var re = ~/-(\[^-]-)/g;
    var s = re.replace('-a-b-c-', '-"$1"-');
    trace(s); // -"a"-b-"c"-
  }
}
```

Another common situation is to switch the position of terms inside a string, as shown in the following code. The expressions look for two words separated by a space and positioned at the end of the string.

```javascript
class Main {
  public static function main() {
    var re = ~/([^\s]+)\s(\w+)$/g;
    var s = re.replace('Name: John Doe', '$2, $1');
    trace(s); // Name: Doe, John
  }
}
```

Patterns

In this section, the fundamentals of the regular expressions patterns are described. Some advanced features have been omitted because they are beyond the scope of the book, are not fully cross-platform (because of differences in the native-platform implementation), or are rarely used.

Characters

Patterns are composed by sequences of characters. Each character in the pattern matches itself in the tested string, so if the pattern is the sequence `abc`, the tested string is matched if it contains the very same sequence. The string passed as an argument to the `match()` method is the test string. The pattern is compared to that string character by character always in forward direction. Every character that is matched against the pattern is “consumed” and it is no more reused in the match; there are exceptions to this rule as explained later on. If one correspondence in the test string is found, the match process is interrupted and the method returns `true`; if the last character of the test string has been examined and no matches have been found, the function returns `false`. 
Chapter 8: Cross Platform Tools

Some characters assume a special meaning when positioned in a specific position of the pattern (see Table 8-6); when you have to match the symbol and not use its special meaning, you have to escape the character with the backslash \ prefix. Note that if you use the regex special syntax, the slash character / must also be properly escaped; this character has no special meanings in the pattern definition but it is used as a terminator for the regular expression.

Table 8-6

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved characters: ./*?^$(){-</td>
<td>Those are the characters that most commonly need to be escaped when you have to match the symbol and not use its special meaning. There are situations when escaping is not necessary because the regex parser can easily disambiguate the use of the character by the context.</td>
</tr>
<tr>
<td>any reserved character between \Q and \E</td>
<td>Suppresses the meaning of special symbols. Does not work in JavaScript.</td>
</tr>
<tr>
<td>\x00</td>
<td>Where 00 is any valid combination representing an ASCII/ANSI character expressed as a hexadecimal value.</td>
</tr>
<tr>
<td>\n \r \t</td>
<td>Matches a newline, a carriage return, and a tab character respectively.</td>
</tr>
</tbody>
</table>

If you instantiate an EReg object using the standard constructor, you may find confounding the use of the backslashes because they are both used to escape the special characters in the regex patterns and to escape the string literal values as described in “The Simple Value Types” section of Chapter 3. The backslash in the example has no special meanings in the regex pattern and it is used only to create a valid string literal.

```javascript
var re = new EReg("a\"b", "i");
```

Character Classes and Sets

Matching for only one character is useful but often limited; character sets and character classes are a way to instantly define wide ranges of symbols (see Table 8-7). A character class is a user-defined set of symbols; the pattern is a match if at least one character in the set matches the current character in the string. A character class is defined enclosing a string sequence inside the square parentheses. So the sequence [abc] means any character that is an “a,” “b,” or a “c.” A character set is a regex hard-coded shortcut to a character class; the character set \d is equivalent to the character class [0-9] and it means any digit character.
Part II: Server Side, JavaScript, and Flash; Oh My!

Table 8-7

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Used as the first character in a character class, it negates the expression; otherwise, it acts normally. So this pattern <code>[^ab]</code> means any character that is not an &quot;a&quot; or a &quot;b,&quot; but this <code>[^^]</code> means that only the characters &quot;^&quot; and &quot;^&quot; are a valid match.</td>
</tr>
<tr>
<td>-</td>
<td>Used inside a character class and between two other characters, it defines a range; used as the first character of or outside a character class, it acts normally. The pattern <code>[a-e]</code> means any character between &quot;a&quot; and &quot;e&quot; included, <code>[.-/]</code> means one of “.”, “.” or “/”.</td>
</tr>
<tr>
<td>\d</td>
<td>A character set for every numeric character (digit). Equivalent to <code>[0-9]</code>.</td>
</tr>
<tr>
<td>\w</td>
<td>Any word character; a word character is any letter or number. Equivalent to <code>[a-zA-Z0-9_]</code> (note that the underscore character is included in the word characters).</td>
</tr>
<tr>
<td>\s</td>
<td>Matches any whitespace character; whitespace characters are the single space (ASCII 32), the tab (ASCII 9), the newline (ASCII 10), and the carriage return (ASCII 13).</td>
</tr>
<tr>
<td>\D</td>
<td>Any non-digit (\d) character.</td>
</tr>
<tr>
<td>\W</td>
<td>Any non-word (\w) character.</td>
</tr>
<tr>
<td>\S</td>
<td>Any non-whitespace (\s) character.</td>
</tr>
<tr>
<td>.</td>
<td>Any character but carriage return (\r) or newline (\n). With the s option in the EReg constructor also allows to match \r and \n. The pattern j.n means any sequence that contains a “j” followed by any two characters and followed by an “n”; so that “john” and “jean” match the pattern but not “jane.”</td>
</tr>
</tbody>
</table>

In the following example, a character class is used to define a pattern that matches any phrase that contains one vowel:

```javascript
class Main
{
    public static function main()
    {
        var re = ~/[aeiou]/;
        trace(re.match('haXe')); // true
        trace(re.match('xxx'));  // false
    }
}
```

In the character class the dash symbol (–) can be used to create ranges of values. In this case, the pattern matches any phrase that contains two hexadecimals in sequence. The pattern is used in case-insensitive mode because of the applied i modifier.

```javascript
class Main
{
    public static function main()
    {
        var re = ~/[0-9a-f][0-9a-f]/i; // same as ~/[0-9a-f]{2}/i;
        trace(re.match('FF'));        // true
    }
}
```
Character sets avoid repeating the same definition many times. In this case, a sequence corresponding to a numerical expression for a date is used as expression. The \W represents any non-alphanumeric character.

```php
class Main {
    public static function main()
    {
        var re = ~/\d{4}\W\d{2}\W\d{2}/;
        trace(re.match('My birthdate is 1972-05-07')); // true
        trace(re.match('My birthdate is 1972/05/07')); // true
        trace(re.match('My birthdate is 1972 05 07')); // true
        trace(re.match('My birthdate is 1972 May 07')); // false
    }
}
```

**Anchors**

Anchors match positions instead of characters; for this reason they do not consume characters (see Table 8-8).

**Table 8-8**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^ (caret)</td>
<td>Matches at the beginning of a string or after a newline. With the m option it matches only the beginning of the string</td>
</tr>
<tr>
<td>$ (dollar)</td>
<td>Matches the end of a string or after a newline. With the m option it matches only the end of the string</td>
</tr>
<tr>
<td>\b</td>
<td>Matches a word boundary. A word boundary can be the start or the end of a sequence of \w characters.</td>
</tr>
</tbody>
</table>

The following pattern matches any string that begins with abc; the rest of the string has no consequences for the match.

```php
class Main {
    public static function main()
    {
        var re = /^abc/;
        trace(re.match('abcdef')); // true
        trace(re.match('abcdef')); // true
        trace(re.match('defabc')); // false
    }
}
```
Part II: Server Side, JavaScript, and Flash; Oh My!

It is possible to check both the limits of a test string. In this case the string is a match only if it contains alphanumeric characters (or the underscore symbol) and no other characters. The second example fails because there is a whitespace at the end.

```javascript
class Main {
    public static function main() {
        var re = ~/\w+/i;
        trace(re.match('John')); // true
        trace(re.match('John ')); // false
    }
}
```

Quantifiers

Characters, character classes, character sets, or groups can be repeated in a pattern using the quantifiers. This avoids error-prone repetitions, pattern bloating, and of course gives the opportunity to set optional repetitions or limit the repetitions to a range of values. The quantifiers described in Table 8-9 always follow a pattern expression that can be a single character or a more complex group.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>The expression is optional.</td>
</tr>
<tr>
<td>*</td>
<td>The expression can be repeated without limits or be absent.</td>
</tr>
<tr>
<td>+</td>
<td>The expression must be present and can be repeated several times.</td>
</tr>
<tr>
<td>{n}</td>
<td>Where n is a positive integer value. The expression must exist exactly n times.</td>
</tr>
<tr>
<td>{n,m}</td>
<td>Where n and m are positive integer valued and m is greater than or equal to n. The expression must exist between n and m times.</td>
</tr>
<tr>
<td>{n,}</td>
<td>Where n is a positive integer value. The expression must exist at least n times.</td>
</tr>
</tbody>
</table>

In the following example, the pattern matches any occurrence of the character j followed by any kind of character and by the n character:

```javascript
class Main {
    public static function main() {
        var re = ~/\[j.\+]n/\;\i;
        trace(re.match('John'));       // true
        trace(re.matched(1));          // John
        trace(re.match('John and Jane')); // true
        trace(re.matched(1));          // John and Jan
    }
}
```
The first match is easy to understand but the second requires an explanation; someone may have expected that the matched value of the second \texttt{match()} could be \texttt{John} and not \texttt{John and Jan}. To understand what has happened you have to know that the regular expression engines are said to be eager. Because the \texttt{.} character matches any character, the first occurrence of the \texttt{n} in \texttt{John} is consumed by the dot while the one in \texttt{Jane} must necessarily be consumed by the \texttt{n} pattern for a match to occur.

How do you prevent this behavior? By changing the way the engine works and making it \textit{lazy}. Any quantifier can be suffixed with an optional question mark \texttt{?} symbol. If present, the expression is processed in lazy mode. That means that after the \texttt{j} has been encountered; the following dot expression is applied as little as possible. In this case the engine tries to apply it just once and matches the \texttt{o} of \texttt{John} and tries to apply the next expression \texttt{n} to the following character \texttt{h}. Because there is no correspondence, the engine rolls back and tries again to apply the dot expression on the \texttt{h} character obtaining a success. This time the next expression \texttt{n} matches correctly and the engine ends its process successfully.

```php
class Main {
    public static function main() {
        var re = ~/(j.+?n)/i;
        trace(re.match('John and Jane')); // true
        trace(re.matched(1)); // John
    }
}
```

### Grouping, Alternation, and Back-references

A pattern can include groups. Any matched group can be extracted using the function \texttt{matchedGroup()} with the numeric id of the group as argument (Table 8-10). The group with id 0 corresponds to the whole match while the groups starting from one are user defined using the standard parentheses. The pattern \texttt{((.)(\d{2}))} defines three groups; the first matches one character followed by two digits, the second is the first character of the previous group, and the third the two digits.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{?:pattern}</td>
<td>Defines a non-capturing group. This kind of group cannot be retrieved using back-references or matched groups and it is useful for grouping alternations.</td>
</tr>
<tr>
<td>\1 to \9</td>
<td>A back-reference is a numeric value prefixed with the backslash character. A back-reference corresponds to a matched group. So the expression \texttt{([ab])x\l} matches \texttt{axa} and \texttt{bxb} but not \texttt{axb} or \texttt{bxa}.</td>
</tr>
<tr>
<td></td>
<td>The pipe character is used to create an alternation. The symbol breaks a pattern into two parts: the string on the left of it and the string on the right; if one of the two is a valid pattern for the tested string, the pattern is a match. Alternations can also be used inside groups. The expression \texttt{j(ohn</td>
</tr>
</tbody>
</table>
Part II: Server Side, JavaScript, and Flash; Oh My!

Back-references are used to make assumptions on already parsed portions of the test string. In this case, the pattern defines that the test string must begin with a vowel, be followed by any number and type of characters, and be concluded with the same vowel encountered at the beginning (case-insensitive because of the i option).

```javascript
class Main {
    public static function main() {
        var re = ~/^([aeiou]).*\1$/i;
        trace(re.match('area')); // true
        trace(re.match('are'));  // false
    }
}
```

Alternation has the lowest priority and, thus, can be used in groups. In the following example, a non-capturing group has been used because there is no interest in retrieving the matched portion defined by the alternation.

```javascript
class Main {
    public static function main() {
        var re = ~/j(?:oh|ea)n/i;
        trace(re.match('John'));  // true
        trace(re.match('Jean'));  // true
    }
}
```

### Look-around, Conditionals, and Comments

Look-around constructs permit you to verify that the pattern is preceded or followed by an expression without including that expression in the match. That means that it is possible to check that the pattern is followed by a certain condition without including that condition in the result. In practice the look-around expressions do not consume characters (see Table 8-11).

The following pattern looks for any occurrence of the word John that is followed by a space and Doe but only Joe is considered a match. If the pattern continued after the look-ahead expression, it would resume exactly at the whitespace character that follows John.

```javascript
class Main {
    public static function main() {
        var re = ~/John(?= Doe)/;
        trace(re.match('My name is John Doe')); // true
        trace(re.matched(0));                    // "John" and not "John Doe"
        trace(re.match('My name is John Roe')); // false
    }
}
```
You can find examples for other look-around expressions in the following code. The last two matches do not work in JavaScript because look-behind is not supported on this platform.

```
class Main {
  public static function main() {
    trace(~/John(?! Doe)/.match('My name is John Doe'));  // false
    trace(~/John(?! Doe)/.match('My name is John Roe'));  // true
    trace(~/(?!John )Doe/.match('My name is John Doe'));  // true
    trace(~/(?!John )Doe/.match('My name is John Roe'));  // false
  }
}
```

### Unicode Support

All three platforms support patterns that contain Unicode multibyte characters (see Table 8-12). On Neko, the regex support is provided by the PCRE library that must be compiled with the Unicode support; the Windows setup, starting from haXe version 1.16, adopts the Unicode support by default.
Timer

The main functionality of the haxe.Timer class is to permit the repetition of the execution of a function over time. A timed execution is specifically addressed to event-based environments such as Flash and JavaScript and, thus, Neko is excluded from the supported platforms. The only function that is also available in Neko is the static method Timer.stamp() that returns the current time in seconds.

The Timer class requires that an interval time expressed in milliseconds, is set in the constructor. The timer activates immediately upon the object instantiation and executes the run() method on each elapsed interval. By default the run() method does nothing and the developer must replace the empty definition with his or her own implementation. The stop() method interrupts the repetitions. Note that once interrupted the execution cannot be resumed and a new Timer instance must be created in place of the previous one.

In the following example, a timer instance is created. The run() method is implemented and traces a counter value. On the tenth repetition the timer is stopped.

```haxe
class Main
{
    public static function main()
    {
        #if !neko
            var timer = new haxe.Timer(100); // 0.1 seconds
            var counter = 1;
            timer.run = function()
            {
                trace("count: " + counter++);
                if(counter > 10)
                    timer.stop();
            }
        #end
    }
}
```

Table 8-12

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\u{0000}</td>
<td>Where 0000 is a hexadecimal value for a Unicode character. Only supported on JavaScript.</td>
</tr>
<tr>
<td>\x{0000}</td>
<td>Where 0000 is a hexadecimal value for a Unicode character. Supported on Flash 9 and on Neko when the option u is active.</td>
</tr>
<tr>
<td>Unicode symbols</td>
<td>The Unicode characters are directly embedded in the source code or loaded from a text file; both must be encoded as UTF-8 to work properly. Supported on any platform.</td>
</tr>
</tbody>
</table>
Chapter 8: Cross Platform Tools

Delayed Action

The Timer class also permits to easily delay the execution of a function that is not repeated. To do this, use the Timer.delayed() static method. Note that the method returns a function; once this is executed the timer starts and the delayed action is performed after the set interval.

```haxe
class Main
{
    private static function delayedAction()
    {
        trace("delayed trace");
    }
    public static function main()
    {
        #if !neko
            var delayed = haxe.Timer.delayed(delayedAction, 200);
            delayed();
            trace("I trace before the delayed once");
        #end
    }
}
```

Queued Actions

Finally the static method queue permits creating a stack of functions; each function can have a different interval before execution, even zero that is the default value if the interval is omitted, and it is removed from the queue as soon as it is invoked. This method can be useful to create sequenced effects. In the following example, the print function is executed with different parameters using the callback statement and at different time delays.

```haxe
class Main
{
    private static function print(s)
    {
        trace(s);
    }
    public static function main()
    {
        #if !neko
            haxe.Timer.queue(callback(print, "One ...."), 500);
            haxe.Timer.queue(callback(print, "Two ...") , 1000);
            haxe.Timer.queue(callback(print, "Three ...") , 2000);
        #end
    }
}
Part II: Server Side, JavaScript, and Flash; Oh My!

MD5

The Message-Digest 5 (MD5) is a cryptographic algorithm intended to generate a hash value of 128-bit length. This very big number is usually represented using a sequence of 32 hexadecimal digits. Given the same input, a cryptographic hash function like the MD5, always returns the same output but it is not possible to reverse the process and to obtain the input directly from the output. The process is not 100 percent secure because there are techniques that can guess the input adopting some comparison techniques of inputs and outputs, but for the more common usages it is practical, fast, and safe enough and it is always possible to adopt complementary techniques to strengthen the process.

Cryptographic hashes are commonly used as digital fingerprints to identify file binaries and securely store user passwords. The first most common case is to ensure that a transmitted file has been received untempered or uncorrupted; on the source of the hash the file is produced and distributed along with the binary file. A client receives the file and makes its own hash encoding; if the result matches with the original hash the file has been received unchanged.

The other very common use in user access management is to store the user password cryptographic hashes instead of the real password; needless to say, it can be dangerous having all the user passwords stored in one place and bi-direction cryptography is only a small step better. Each time a user attempts a log in, the provided password is encoded in its hash form and matched against the hash stored in the users' database.

The haxe.Md5 class provides just one single public static method encode() that accepts a single string argument, the value to encode, and returns a string result, the hash.

```haxe
class Main
{
    public static function main()
    {
        var hash = haxe.Md5.encode("myverysecretpassword");
        trace(hash);
        // the result is consistent on all platforms and it is
        // eb18a77e09d2eaa6896637e73f4ff94f
    }
}
```

Summary

In this chapter, you saw four fundamental tools that are cross-platform and very commonly used. In the rest of the book those tools are used a lot and you will probably want to refer to this chapter several times. The topics of the chapter have been:

- XML parsing and the XML API
- EReg API and regular expression patterns
- Temporized and delayed executions using the Timer class
- Creating cryptographic hashes of passwords and other strings using the MD5 class

In the next chapter you learn how to build sophisticated and dynamic websites using Neko.
The haXe development environment was born with the main purpose of developing web applications. A web application is composed of several components, both client- and server-side. The client aspect can be resolved with haXe targeting Flash or JavaScript whereas Neko manages the server-side. In this chapter you will take a deep look at building websites using Neko by covering the following topics:

- What a web server is
- What HTML is
- How to write HTML pages
- What HTML elements are used
- How to use the NekoTools as a development web server
- How to install mod_neko for Apache
- How to generate dynamic pages with Neko

**Introduction to Web Development**

Web programming is a branch of programming that targets the realization of software deployed to work on the Internet. This kind of software requires the participation of at least two parts: a client that sends a request and a server that provides a response. The web programmer writes a code that runs on the server, processing data obtained by a database for example, and sends a response. The response can be anything such as a text page, a downloadable archive, or a piece of code to run into the client. To write a web application you need to become confident with what a web server is and how it works, and how to write HTML pages, which actually are the very base for producing interactive contents.
What Is a Web Server?

A web server is both an application and a personal computer (PC) configured to serve web contents. A web server application is an executable that resides in the PC memory and waits for requests on the HTTP (Hypertext Transfer Protocol) communication channel. When such a request is received, the web server analyzes the input and tries to produce a suitable response; this can be transmitting an existing file available on the server, such as an image or a static document, or delegating the execution of some script/bytecode to an interpreter and sending back to the client the result of such an operation. The most traditional client for the HTTP protocol is the web browser, although it surely is not the only one. From now on, client software will be referred as user agents. They can span from modern Internet browsers such as Firefox, Safari, or Internet Explorer to search engine robots harvesting information to collect.

When a client submits a request for a page to a server, it starts a flow of information that concludes in the transmission of HTML (Hypertext Markup Language); the steps to obtain an answer from the server are as follows:

- The Internet browser receives from the user an HTTP address of the desired resource. This address can be directly inserted or be a hyperlink in an existing page.
- The user agent submits the desired URI (Unified Resource Identifier) along with other information over the Internet connection; the request is sent from server to server until it reaches the machine that can address the correct response.
- The server processes the request and sends a response containing the HTML code.
- The browser begins to receive the HTML code and starts the rendering process as soon as it receives an entire block of information. A block can be a paragraph, an image reference, or a table. That piece of code is interpreted and transformed into a visual representation. This is the reason why sometimes pages seem to compose one piece at the time when the transmission is not particularly efficient or the page content is quite heavy.
- When an embedded object, such as an image, a script, a flash movie, or a style sheet is encountered during the page rendering, a new request is submitted to the server and the result included in the page. This is an asynchronous process meaning that the embedded assets may be fully loaded before or after the HTML code has been completely received and/or processed.

The address is not the only information sent to the server in the request; other information (usually hidden to the end user) is packed in the request in the form of headers. These headers may contain glitches about the identification of the browser, the preferred languages, or the contents of a form in the page. In the same way the response may contain some extra information in the response header, like the content type associated with the transmitted information or some values to store in the client cookies (a small portion of the client memory reserved to store preferences for the current navigation context).

Distributing Contents Using a Web Server

To understand how a Web Server can send contents over the HTTP protocol, you need to understand how an HTTP address is structured. The address is composed of the HTTP prefix, a server name, like www.haxe.org, and an optional relative path to a specific resource located on the server. Certain web applications also accept other protocols like HTTPS (same as HTTP but over a secure connection) or FTP (File Transfer Protocol). The name of a server can be composed of just one name when it is a local
address or, when it is publicly accessible and registered in the DNS (Domain Name Server) service, of
two or more names separated by a dot. A server can also be reached using its numeric address composed
of four dot-separated numbers between 0 and 255. In most configurations, the two following addresses
are equivalent:

http://localhost/
http://127.0.0.1/

A site available on the Internet has a name composed of domains with the most generic one on the right.
The address http://www.haxe.org/ is composed by several parts, starting from the right side:

- org is a top-level domain (TLD) reserved to host sites of organizations with no economic
  interests. Dedicated organizations maintain the top-level domains and cannot be acquired by
  individuals or companies. The most common TLDs are:
  - com for commercial sites
  - net for generic network sites
  - edu for educational institutes and entities
  - gov for U.S. Government institutions
  - The many regional domains like.fr for France and it for Italy
- haxe is a first-level domain. It is an arbitrary name acquired for a limited time by an individual
  or a company. The domain subscription is usually renewed on a per-year base.
- www is a second-level domain. A first-level domain owner can create as many second-level
  domains as desired; some hosting companies cover a price for second-level domain related to
  technical costs. The www second-level domain is nowadays a de facto standard to locate the main
  entry point for most existing first-level domains. The presence of an SLD is optional.
- http is the protocol used to connect to the resource specified by the address.

A web server is usually configured so that its root address points to the content of a physical directory
on the hosting machine. Thus, if a web server, say example.com, is configured to have its root to point to
the directory D:\www (or //path/to/www on *nix) that contains a file named page.html, this resource
can be reached on the Internet using the URI address http://www.example.com/page.html.

Directories and subdirectories are mapped one to one separated by the slash symbol / . When a URI
points directly to a directory and not to a file (may it be virtual or real), the web server can show the
content of the folder if so configured, or redirect to a default page. The NekoTools redirects the request to
the first file encountered in the directory between index.html, index.htm, and index.n. If none of
those options are available, an error Page Not Found (code 404) is reported. Production web servers such
as Apache have several options to deal with files, folders, and virtual paths, restricting accesses and so on.

In many cases, the server assembles the page contents dynamically, varying their contents with the
parameters received in the request. Those parameters can be transmitted using the GET method or the
POST method (there are also some other less common methods). The first appends the parameters to the
address while the second uses the headers of the request to store them. The syntax to append parameters
using GET is the following:

http://www.example.com/page.aspx?param1=value1&param2=value2
Part II: Server Side, JavaScript, and Flash; Oh My!

The param1 is a variable name while value1 is its value. The parameters are separated from the address part by a question mark symbol and between them by the ampersand symbol (&). Each key-value pair is composed by the variable name, the equal symbol (=) and the value. Because the HTTP restricts the use of some characters in the URI, values and parameters must be encoded.

While the GET method is implicitly used for every standard hyperlink on a page, to use the POST method it is necessary to recur to a form submission specifying the desired method. Another common way to make POST requests is to use an XHR (XML HTTP Request) call that is the very base of AJAX (Asynchronous JavaScript and XML) applications. The POST method does not use the query string to pass the parameters; it encapsulates them in the request headers.

The protocol does not impose any size limit on the GET parameters but some (older) browsers and web servers truncate the query string to a fixed (quite long) size. For this reason as a general rule of thumb, the GET method is used with short, user-friendly parameters, while the POST is used for complex, potentially very heavy content submissions. Finally, file uploads can be accomplished only using the POST method.

So far, URI addresses have been used in their absolute form, a full path that includes the server address. The relative form is also admitted in many contexts. A relative path is used to point to a resource in relation to the current one or to a base one. A relative path does not specify a protocol because the current context is used. The path is a combination of subdirectory names and a filename that, used in conjunction with the current document address, directs to the desired resource. The path can also be navigated backward, from child to parent, using two dots in place of a directory name. A relative path starting with a slash symbol (/) indicates that the URI is relative to the current host address and not to the current address.

Thus, if the current page URI is http://www.example.com/pages/first.htm the relative path second.htm will point to the address http://www.example.com/pages/second.htm. The relative path /images/logo.png will point to http://www.example.com/images/logo.png. The same URI can be defined as the relative path../images/logo.png.

An HTML Crash Course

HTML is a language describing the structure of a text. Each HTML file describes a single page that can contain references (hyperlinks) to other pages. The language is based on markups, technically known as tags, which are keywords enclosed in angular brackets. Two tags, one opening and one closing, delimit the elements that can have contents (text and/or sub-elements), whereas the elements that contain nothing are described using the empty tag. The keyword in the tag is known as Element Name.

```html
<div>content</div>
```

In the preceding example, the plain text content is included in the element with name div. The div element is used to define blocks inside the page and is one of the most frequent used structures. Note that the end tag replicates exactly the start tag except for the use of the slash symbol (/) just after the opening angular bracket.

```html
<br />
```
In this case, the tag is empty because it does not contain other contents (and can’t really do that) and it is
differentiated by using the slash just before the closing bracket. The br tag adds a line break in the text.
Note that new lines are not the same as paragraphs that must be defined using the p element.

An element can also possess additional information in the form of attributes. Attributes are defined in
the opening tag and in the empty tag in the form of space separated key-value pairs. Each pair is
composed by an attribute name, an equal sign, and a value enclosed in double quotes.

\[<a \text{href="/page.html">link text</a}>\]

The preceding example marks the link text content as a hyperlink to the file page.html in the root of the
current website.

The HTML standard allows for a more flexible syntax that allows, among other things, to:

- Indifferently use upper- and lowercase for element names and attributes.
- Omit certain attribute values.
- Use single quotes for attribute values (or no quotes at all when there is no ambiguity possible).
- Avoid distinguishing between the empty elements syntax and the start-tag syntax (empty
tags do not have to be marked with the slash). The empty-tag format is not part of the HTML
standard at all, whereas it is from XHTML/XML. The browsers are very permissive on this side
and don’t complain about this adoption and many common validators, applications written to
check the correctness of a document, do not consider this variation as an error.
- Certain elements can be implicitly closed (like table cells).

In this chapter, you have chosen for a more XML-oriented syntax because it produces a clearer code,
which is easier to read and maintain and can be used easily in conjunction with tools that work with
XML contents.

**Base HTML Document Structure**

Every document generated in HTML must have a minimum set of elements such as in the
following example:

\[<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    Content Goes Here
  </body>
</html>\]

The first two lines of the document are the Document Type Definition (DTD), which is a special heading
indicating which standard applies to the current document. In this case, the HTML version 4.01 in its
strict flavor has been chosen. Other flavors of the last HTML version as defined by the World Wide Web
Consortium (W3C, www.w3c.org/) are the transitional and frameset ones. The strict flavor has less
defined elements and no deprecated elements and arguments from previous versions in its definition;
also the frameset features have been skipped. This leaves a syntax that is all dedicated to the semantic of
the documents and not to its visual representation. Styles and formatting rules are completely delegated
to style sheets that can be embedded in the document or externally referenced. In this chapter, the HTML
strict flavor will be referred to with a small exception: The conventions used for empty tags and Boolean
attributes are those defined in XHTML.

The Document Type Definition contains an optional web address that contains a resource, a DTD file,
which fully describes the syntax of the current document.

After the DTD, the first element encountered is html; it must be unique in the page and it represents the
root node that will contain every other element in the page. Inside the HTML element the two elements
head and body are declared. The first contains information that is used on the page as a whole; it can be
meta information like the character encoding that applies to the document where the most common is
UTF-8, or the inclusion of external files that defines styles (CSS style sheets) and behaviors (JavaScript).
In the example, it contains the very least mandatory element title. This is a sort of label associated to
the page and it is usually displayed by the browsers in their title bar (it is not shown in the rendered
page because it belongs to the head of the document); it can only contain plain text (text that does not
contain other tags). The other element body is the actual content of the page. The body of the page
can contain a mix of plain text and elements.

The HTML syntax relies on some fixed structures (for example, a cell element must be contained inside a
row element) but you can easily experience that browsers are quite tolerant with wrong structures or
badly written markups. Anyway it is a good practice to validate the code (manually or using one of the
many available tools) to avoid browser inconsistencies in visual rendering. Browsers already behave in
different ways with codes that respect standards.

When you explore the HTML code of many pages in different sites, a very wide adoption of indentation
styles and use of whitespaces can be encountered. This is possible because multiple whitespaces are
treated as one unless a specific style rule exists to say something different. Whitespace characters include
the single space produced pressing the spacebar, tabulation symbols, and newline characters.

Almost every element supports the use of two fundamental attributes: id and class. The value of the
first must be unique within the whole page. It is used to look at a specific element by name. The latter
can have one or more arbitrary string values separated by whitespaces; the same class value can be used
in several elements to aggregate them semantically. They are both widely used in the context of scripting
and styling to point exactly to certain elements.

In Appendix A, you can find a full description of what is defined as “Semantic HTML.” The elements
and attributes covered are all of the HTML 4 specifications, but only the ones that have a semantic
meaning have been treated; deprecated or format-only structures have been completely removed.

**How haXe Differs from HTML**

haXe is a programming language, and its purpose is to describe algorithms and procedures that the PC
should interpret and resolve; HTML is a descriptive language used to create complex text-based
documents. The haXe syntax is interpreted by the compiler and transformed into a machine language
ready to be used in software applications. The HTML syntax is interpreted by a software agent (usually
a browser) and rendered to a media (such as the PC screen, a printer, or an audio reader software).
Their usages and structures are completely different; what they have in common is the final purpose: providing contents and user interfaces for web applications. haXe provides the logic and HTML provides one of the possible outputs.

Neko, introduced in Chapter 1, is a great platform for server-side applications; it is fast and very conservative on memory and CPU demands. It has all the aces to become one of the great players for advanced web applications.

The NekoTools Web Server

The haXe distribution comes with a handy tool named NekoTools. The application has two purposes: transforming a Neko file (.n) into an executable (.exe on Windows) and serving web contents. The latter provides a nice environment to test server applications that target the Neko platform. It is not advisable to use the NekoTools as a production server but only as a lightweight development tool.

> The windows distribution comes with a haxeserver.bat that is a shortcut to start the NekoTools application as a web server.

The NekoTools can be configured using some switches on the execution command line and using a special web page as it will be explained shortly. The base command to use NekoTools as a web server is as follows:

    > nekotools server

Used this way, an instance of the web server is executed and ready to accept calls over the HTTP protocol on the default port 2000.

The NekoTools application can use the optional parameters described in Table 9-1.

### Table 9-1

<table>
<thead>
<tr>
<th>Switch Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-p &lt;port&gt;</td>
<td>Uses a different server port. The default one is 2000 if the switch is omitted.</td>
</tr>
<tr>
<td>-h &lt;host&gt;</td>
<td>Changes the server host. The default value is localhost.</td>
</tr>
<tr>
<td>-d &lt;dir&gt;</td>
<td>Sets a specific directory as the root of the web server.</td>
</tr>
<tr>
<td>-log &lt;file&gt;</td>
<td>Sets a log file to take trace of the operations performed by the web server.</td>
</tr>
<tr>
<td>-rewrite</td>
<td>Activates the URL rewriting functionalities for smart URLs. To activate those functionalities, a file .htaccess must be added in the web server directory. The same syntax as for the mod-rewrite module for the Apache web server has to be applied. With this module it is possible to redirect virtual paths to real ones.</td>
</tr>
</tbody>
</table>
Part II: Server Side, JavaScript, and Flash; Oh My!

The NekoTools provides a magic page reachable at the address http://localhost:2000/server:config (the port and the server may be different if so configured) that can be used to change the current working directory or simply to verify in which directory the web server is running.

Installing mod_neko For Apache

If the NekoTools is ideal for the development environment, Apache is surely a bet when your work must go public. Apache is a very solid and reliable web server capable of responding correctly to many simultaneous requests and with a very small memory and processing footprint. Right now it is also the only possible choice to deploy a Neko application on a server machine although things may rapidly change in the near future.

To use Apache as a web server for processing Neko applications, it will be necessary to install the mod_neko module.

Installing Apache and mod_neko on Windows XP

Open your Internet browser and type the URL http://httpd.apache.org/download.cgi. This is the download page for the Apache web server. Download the latest stable version available for Windows (version 2.2.4 at the moment of writing). You will find it in at least two flavors, Win32 Source and Win32 Binary. The first contains all the resources to compile the server from source whereas the second contains just an installer application ready to use. Choose the second and execute the downloaded file.

A wizard will guide you to the software installation; accept the default settings (you only have to actively select the license agreement condition) and click Next on each screen until the installation process is over.

Now select Apache HTTP Server 2.2.x Configure Apache Server Edit the Apache httpd.conf Configuration File from the Windows Start menu. The Notepad application opens with a configuration file in it. It is the httpd.conf file that contains the main configurations for the Apache web server. Remember that each time you change something in this file, the web server must be restarted in order to let the changes take place.

Now go to the end of the file and add the following lines:

```
#BEGIN NEKO
LoadModule neko_module "C:\Program Files\Motion-Twin\neko\mod_neko2.ndll"
AddHandler neko-handler .n
DirectoryIndex index.n
#END NEKO
```

The path C:\Program Files\... might have to be adjusted for localized versions of windows or when haXe is installed in a custom path.

The first and the last lines are just comments to mark your modification and you can omit them.
The `LoadModule` line just instructs the web server to load the `neko_module` from the specified location. You may have to change the path if you have installed neko in a different directory. The double backslashes in the path are not an error.

*In the Neko directory there is also a mod_neko.ndll (without the 2 before the dot) that is the module for Apache version 1.3.x.*

The following line declares that each request for files ending with the `.n` extension must be processed by the `neko-handler` that is part of the `neko_module`.

Finally the `DirectoryIndex` line suggests trying to look for and serve an `index.n` file when the path to a directory is requested from a user agent.

That’s it!

To test your installation, compile an `.n` file and copy it to the default web server directory (by default `C:\Program Files\Apache Software Foundation\Apache2.2\htdocs`). Now point your browser to the URI `http://localhost/filename.n` where `filename` is the name of your compiled Neko file and see the result. Later in this chapter, a real-world example that you can also test in your Apache environment will be provided.

### Installing Apache and Mod_Neko on Linux (Ubuntu)

Many Linux distributions come with Apache already installed. If it is not your case, open a terminal window and type:

```bash
> sudo apt-get install apache-dev apache
```

*Be sure that the installed version is 2.2.x or greater. Otherwise the mod_neko module may not work properly; consult the [www.haxe.org](http://www.haxe.org) website for possible workarounds.*

Insert your administrator password when prompted. The download and install process will start; just confirm every request of the system.

Once the installation is complete, open the file `/etc/apache/httpd.conf` in your favorite text editor. You can use the following command to do this from your terminal window:

```bash
> sudo gedit /etc/apache/httpd.conf
```

Insert your administration password if required and confirm. Now go to the end of the file and add the following lines:

```diff
    #BEGIN NEKO
    LoadModule neko_module "/usr/lib/neko/mod_neko.ndll"
    AddHandler neko-handler .n
    DirectoryIndex index.n
    #END NEKO
```

The first and the last lines are just comments to mark your modification and you can omit them.
Part II: Server Side, JavaScript, and Flash; Oh My!

The `LoadModule` line just instructs the web server to load the `neko_module` from the specified location. You may have to change the path if you have installed Neko in a different directory.

> In the Neko directory, there is also a `mod_neko.ndll` (without the 2 before the dot) that is the module for Apache version 1.3.x.

The following line declares that each request for files ending with the `.n` extension must be processed by the `neko-handler` that is part of the `neko_module`.

Finally the `DirectoryIndex` line suggests trying to look for and serve an `index.n` file when the path to a directory is requested from a user agent.

Now open and edit the file `/etc/ld.so.conf` and add your path to the Neko install directory (for example, `/usr/lib/neko`). Then run the following command from the terminal:

```
> sudo ldconfig
```

Insert your administrator password if required and confirm. This will render Neko accessible from Apache. Restart Apache.

That’s it!

To test your installation, compile an `.n` file and copy it to the default web server directory (for example, `/var/www`). Now point your browser to the URI `http://localhost/filename.n` where `filename` is the name of your compiled Neko file and see the result. Later in this chapter, a real-world example that you can also test in your Apache environment will be provided.

Note that your mileage may vary slightly because of differences in distributions and versions, but if you work on Linux you probably already know that.

Your First Website with haXe

In a previous section dedicated to the HTML syntax, a situation was described in which the web developer writes by hand the pages that he wants to deploy; those documents are known as static, as they are saved in a file and sent unchanged to the user agent.

Nowadays this is becoming a more and more uncommon situation also for small sites; web designers usually provide the skeletons for web contents but rarely the content itself. Some sort of repository, such as the file system or more commonly a database, usually drives the contents. The repository contains the raw information that is handled by the application and it is embedded in the layout provided by the web designer. This way a common layout can be reused in an entire set of pages and in some cases in the whole website. The process of retrieving and processing contents to be assembled in a visual output is described as generating dynamic contents.

Having the contents stored in a database provides many and useful ways to access the data. It is possible to query the database to return just a subset of the available contents; a homepage for example can visualize just the last ten posted news. The contents from miscellaneous sources can be mixed to provide
richer pages, an author page in a bookstore site can show both his biography and his books; when a new book is added to the repository the author page is automatically updated.

The process of mixing content and layout varies a lot usually recurring on a template engine, argument of the next chapter, or simply by producing the desired HTML programmatically.

Now that all the tools of the trade have been exposed, it is possible to create a real website. There are different approaches to building dynamic websites: You can have as many logical units as many pages there are in the website or have just one logical unit that manages every possible request for dynamic contents. The first approach is commonly known as Page Controller because each page is responsible for its execution, while the latter is known as Front Controller, because the website has a single entry point. Of course there is a middle ground; you can have one Front Controller for every big section of your site, for example, one for the administration section and one for the front-end.

**Using Neko as a Page Controller**

A very basic example could be the classic Hello World!

Create a `helloworld` directory somewhere in your disk and two subdirectories, `web` and `src`. The first directory is the project directory; `web` will contain the compiled code and will be used by the web server as the site root, and `src` will contain the code files.

Inside the `src` directory a file `HelloWorld.hx` with the following code is created:

```hx
class HelloWorld
{
    public static function main()
    {
        var content = '
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
    <title>Hello World!</title>
</head>
<body>
    <h1>Hello World!</h1>
    Current time is <b>' + Date.now() + '</b>
</body>
</html>
';
    neko.Lib.print(content);
}
}
```

The function body does two simple things: It stores the content of the page that has to be visualized in the variable `content`, and it prints the variable `content` using the `neko.Lib.print` method. In a Neko web application context, printing means sending the content to the agent that requested it.
Part II: Server Side, JavaScript, and Flash; Oh My!

In the project directory, the file `HelloWorld.hxml` is added. It is a compilation file and will transform
the source code into a compiled unit. The content of the file is:

```
-cp src
-neko web/hello.n
-main HelloWorld
```

The Class Path switch (-cp) is required because the code is not directly contained in the project directory
but it can be found in the relative path `src`. The `-neko` switch tells the compiler to use Neko as the
compilation target and to put the compiled result in the `hello.n` file inside the `web` directory. Finally the
`-main` switch gives an indication that the `HelloWorld` class contains the main function that is the entry
point of the application.

To compile, just open the command-prompt/console, navigate to the project directory, and execute the
following command:

```
> haxe HelloWorld.hxml
```

This produces the file `hello.n` that can be executed by the web server.

*On a Windows machine, a double-click on an `.hxml` file will execute the same command as in the
previous example.*

To see the result of this small exercise, navigate in the web directory of the project and launch the
command:

```
> nekotools server
```

The Neko web server is now active and ready to receive requests. Open your Internet browser and point
it to the following URI:

```
http://localhost:2000/hello.n
```

The result is a very simple HTML page that reports the hello message in big words.

Now the process of building individual Neko pages can be repeated at will. A single `.hxml` file can
compile many Neko files at once, just repeat as many compile sections separated by the `--next` switch.

The page controller approach works fine for very small sites but on a larger scale it is not very good.
Codes tend to be repeated unnecessarily and the number of files to manage and deploy grows very fast.
The next section will show how to implement a basic Front Controller to solve those problems.

**The neko.Web Class**

The standard libraries contain a very useful class `neko.Web` that provides facilities, in the form of static
methods, to deal with web server related issues. The class eases the approach to deal with complex stuff
like headers, the retrieving of parameters from the client request, and more. Table 9-2 shows the
description of the `neko.Web` static methods and the next section discusses how they can be used in a
real-world example.
### Table 9-2

<table>
<thead>
<tr>
<th>Neko.Web Static Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cacheModule( f : Void -&gt; Void ) : Void</td>
<td>Sets the main entry point function used to handle requests. Setting it to null will disable code caching.</td>
</tr>
<tr>
<td>flush() : Void</td>
<td>Flushes the data to the client. Usually data is buffered by Apache and sent in chunks; this method permits to anticipate the submission to the client.</td>
</tr>
<tr>
<td>getAuthorization() : { user : String, pass : String }</td>
<td>Returns an object containing the basic scheme authorization sent by the client.</td>
</tr>
<tr>
<td>getClientHeader( k : String ) : String</td>
<td>Gets a client header value from the client request.</td>
</tr>
<tr>
<td>getClientHeaders() : List&lt;{header : String, value : String }&gt;</td>
<td>Gets a list of all the client headers in the client request.</td>
</tr>
<tr>
<td>getClientIP() : String</td>
<td>Returns the IP address of the client that is making the request.</td>
</tr>
<tr>
<td>getCookies() : Hash&lt;String&gt;</td>
<td>Returns a hashtable containing all the cookies sent by the client. To set a new cookie, don’t change this hashtable and use the setCookie() method instead.</td>
</tr>
<tr>
<td>getCwd() : String</td>
<td>Returns the current working directory in the local filesystem.</td>
</tr>
<tr>
<td>getHostName() : String</td>
<td>Returns the server name of the local host.</td>
</tr>
<tr>
<td>getMultipart( maxSize : Int ) : Hash&lt;String&gt;</td>
<td>Returns the POST data in a hashtable. The data passed to the hashtable cannot exceed the maxSize value or an exception is thrown.</td>
</tr>
<tr>
<td>getParams() : Hash&lt;String&gt;</td>
<td>Returns a hashtable containing the GET and POST parameters.</td>
</tr>
<tr>
<td>getParamsString() : String</td>
<td>Returns the GET parameters in string format the same way as they appear in the URL of the request. Parameters in GET or POST can be expressed as an array of values; this function permits to return the whole array selecting the parameter by name. Parameters must be expressed in the following form: ( p1[]=abc&amp;p1[]=123 ). The returned array for the param ( p1 ) contains two string elements “abc” and “123”.</td>
</tr>
<tr>
<td>getParamValues( param : String ) : Array&lt;String&gt;</td>
<td>Same as the getParamString() but for POST data. Note that the string length is limited to 256KB unless the request is encoded as multipart/form-data; in that case, use the getMultipart() or parseMulti-part() to retrieve the POST values.</td>
</tr>
<tr>
<td>postData() : String</td>
<td></td>
</tr>
</tbody>
</table>
Part II: Server Side, JavaScript, and Flash; Oh My!

<table>
<thead>
<tr>
<th>Neko.Web Static Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getURI() : String</td>
<td>Returns the URI from the original request.</td>
</tr>
<tr>
<td>parseMultipart( onPart : String -&gt; String -&gt; Void, onData : String -&gt; Int -&gt; Int -&gt; Void ) : Void</td>
<td>Parses the multipart data of a POST request. The passed function onPart() is invoked when a new part is found with the part name and the filename as arguments. The argument function onData() is invoked when some data is received that is passed as first argument, its position in the stream, and its length. This method is commonly used to store on the server disk the data received from an upload operation.</td>
</tr>
<tr>
<td>redirect( url : String ) : String</td>
<td>Redirects the request to the passed URL using the header “Location”.</td>
</tr>
<tr>
<td>setCookie( k : String, v : String )</td>
<td>Sets a new cookie with passed key and value. The same restriction explained for setHeader() applies here, too.</td>
</tr>
<tr>
<td>setHeader( h : String, v : String ) : Void</td>
<td>Adds a key-value pair of header information. If something has already been printed to the output and the method is invoked, an exception will be raised.</td>
</tr>
<tr>
<td>setReturnCode( r : Int ) : Void</td>
<td>Set the HTTP return code. The same restriction explained for setHeader() applies here, too.</td>
</tr>
</tbody>
</table>

The class also provides a static variable isModNeko(default,null) : Bool that identifies if the web server is working behind Apache (using the mod_neko library) or not.

A special note must be made about the getCwd() method because it is a common subject of confusion. First, the method returns a path in the local filesystem and not any kind of Internet URI. Second and more important, the path is absolute and indicates the directory location where the neko module has been activated. If you are using NekoTools server, it is the directory where the command has been executed; if you are using the mod_neko module, then it is the base directory configured in the web server; of course the two can coincide in many situations.

**Using Neko as a Front Controller**

The following example is more complex but far more satisfactory. You will pretend to build a basic Wiki application: a site whose contents can be quickly modified by the users directly in their browsers. The word *wiki* derives from the Hawaiian Wiki-Wiki expression, which means *rapid*. The features of the Wiki you pretend to implement are:

- Every page can be immediately modified clicking on an edit link. The edit page is composed by a simple form that allows content alterations.
- New pages can be added simply by navigating to the pretend location and clicking the edit button.
Chapter 9: Building Websites with HaXe

- The syntax of the page content will be a standard HTML fragment (only the elements that can be contained inside the body are permitted).
- Support for a multilevel tree structure. Each page can have unlimited child pages.
- The current page breadcrumbs is automatically generated. In web development, a breadcrumb is an indication of the page location inside a complex structure. When you use the breadcrumbs, it is possible to navigate to the parent containers of the current page.
- The title of each page is derived from its URI.
- Use text files to store the created pages.

The result of this small project is visible in Figures 9-1 and 9-2.

![Figure 9-1](image-url)
Part II: Server Side, JavaScript, and Flash; Oh My!

As already done before, the project structure can be prepared creating a new directory `wiki`. Inside the `web`, `src`, and `pages` directory are created. The `pages` directory will contain the contents of the Wiki that are created dynamically. For this reason, be sure that this directory has the correct permissions to allow Neko to write in it.

The whole code needed to run the application will go in the `src` directory. The entry class is `WikiApplication` and its definition is set in the `WikiApplication.hx` file.

```haxe
import WikiController;
class WikiApplication
{
    public static function main()
    {
        var uri = neko.Web.getURI();
        var repositoryPath = neko.Sys.getCwd() + "../pages";
        var params = neko.Web.getParams();
        var action = switch(params.get("action"))
```

Figure 9-2
The class contains just the main method that will be invoked every time that the web server calls the Wiki application. To produce a valid response, some information must be acquired; first thing first, the URI of the current requested page must be known so that it will be possible to load the matching content. Remember that all the requests will be redirected to your WikiApplication. Later in this section it will be explained how to instruct the web server to redirect all the page calls to the same execution unit. The neko.Web.getUri() method returns a relative path to the current page, so that if the invoked URI is http://localhost:2000/mydirectory/mypage?param=value the value returned will be /mydirectory/mypage

If the address is the web server base address http://localhost:2000/ the returned value will be a simple slash symbol /.

Other information required by the application is the directory where the actual contents will be saved. Right now the path is relative to the current working directory, but you can easily change it to fit your needs, maybe loading this value from a configuration file.

Finally, the application must be aware of every parameter that is associated to the request done to the web server. The neko.Web.getParams() returns a hash table containing all the GET and POST variables. The parameters needed to process the request are the kind of action to perform (view, edit, or save) and the content to save in case the action is “save”. When a request is done without an action parameter or with an action parameter with an invalid value, the view value is assumed by default.

The three actions are very intuitive: view will show the page content, edit will show a form to edit the page content whereas save will store the content and show the newly modified page with a save-confirmation message.

When you have to use a request, always assume that the sent information can be manipulated for malicious intent; transforming the input parameters into an enum is a sure way to prevent errors and unexpected results.

The WikiController.hx, located in the src directory, contains the definitions of the WikiController class and of the RequestAction enumeration. Every text content has been assigned to a private static variable. This is a good practice that makes life easier to the developer when he or she needs to change
some values at a later time because he won’t have to scroll the whole code to find the spots to change possibly forgetting some parts.

In the class constructor, the function arguments are stored in instance variable for later use. In the case of URI, the value is changed into the special value /root when the requested URI is the base root address. In this way, you get a reference name root also for the homepage that can now be treated as any other dynamically created page.

As you have probably noticed, in the main method of the WikiApplication class, the WikiController is instantiated and then the method execute() is invoked. It is in this method that the real action takes place. The execute method decides which view must be rendered and instantiates the relative class. All the classes that actually produce the real output, the HTML code, are children of the abstract class Page and, thus, they all share the same method render(), which finally produces the desired result. The result is sent to the user agent that requested it by using the neko.Lib.print() method. In the case of the save action, the execute() method is also responsible for calling the savePage() or removePage() methods. A removePage() occurs when the user sends an empty content. The file is removed and not simply left empty, to avoid filling the repository of garbage files.

The getPageContent() is a public method used by the Page classes to retrieve the content of the page. A default value may be provided as argument for the method; this is used when the requested page does not exist. The getTitle() and the getBreadcrumbLinks() are also used by the Page classes and they respectively return the title of the page derived by its URI and a list of objects containing the link information about the current page and its ancestors. The other private method exists to support the previous described operations and are quite self-explanatory.

```haxe
import haxe.Stack;
import neko.FileSystem;
import neko.io.File;
import neko.io.FileOutput;
import neko.Lib;
import Page;
class WikiController
{
    private static var FILE_EXTENSION = '.wiki';
    private static var ROOT_PAGE = '/root';
    private static var ROOT_URI = '/';
    private static var DEFAULT_EDIT_TEXT = '';
    private static var DEFAULT_VIEW_TEXT = 'This page does not exist,
    click on edit to create it.';
    private static var SAVE_MESSAGE = 'The page content has been
    successfully saved.';
    private static var HOME_TITLE = 'Home Page';
    public var uri(default, null) : String;
    private var dir : String;
    private var action : RequestAction;
    public function new(uri : String, dir : String, action : RequestAction)
    {
        if(uri == ROOT_URI)
        { uri = ROOT_PAGE;
            if(uri.substr(uri.length - ROOT_URI.length) == ROOT_URI)
            { uri = uri.substr(0, uri.length - ROOT_URI.length);
                this.uri = uri;
            }
        }
    }
}```
Chapter 9: Building Websites with HaXe

```java
this.dir = dir;
this.action = action;
}
public function execute() : Void
{
    var page : Page;
    switch(action)
    {
    case Edit:
        page = new PageEdit(this, DEFAULT_EDIT_TEXT);
    case View:
        page = new PageView(this, DEFAULT_VIEW_TEXT);
    case Save(content):
        if(content == '')
            removePage();
        else
            savePage(content);
        page = new PageView(this, null, SAVE_MESSAGE);
    }
    Lib.print(page.render());
}
public function getPageContent(alternative : String) : String
{
    if(pageExists())
        return neko.io.File.getContent(getPageFile());
    else
        return alternative;
}
public function getTitle() : String
{
    if(uri == ROOT_PAGE)
        return HOME_TITLE;
    else
        return StringTools.urlDecode(uri.substr(uri.lastIndexOf('/', 0) + 1));
}
public function getBreadcrumbLinks()
{
    var list = new Array<LinkItem>();
    if(uri != ROOT_PAGE)
    {
        var path = getPageFile();
        while(path.length > dir.length)
        {
            if(FileSystem.exists(path))
                list.unshift({ title : titleFromPath(path), uri : uriFromPath(path) });
            else
                list.unshift({ title : titleFromPath(path), uri : null });
            path = path.substr(0, path.lastIndexOf('/'));
            if(path == dir)
                break;
            path += FILE_EXTENSION;
        }
    }
}
```

(continued)
list.unshift({ title : HOME_TITLE, uri : ROOT_URI });
return list;
}
private function getPageFile() : String
{
    return dir + uri + FILE_EXTENSION;
}
private function getPageDirectory() : String
{
    return dir + getPageNamespace();
}
private function getPageNamespace() : String
{
    return uri.substr(0, uri.lastIndexOf('/'));
}
private function pageExists() : Bool
{
    return neko.FileSystem.exists(getPageFile());
}
private function uriFromPath(path : String)
{
    var relative = path.substr(dir.length);
    return relative.substr(0, relative.length - FILE_EXTENSION.length);
}
private function titleFromPath(path : String)
{
    var file = StringTools.urlDecode(path.substr(path.lastIndexOf('/')+1));
    return
        if(file.substr(file.length - FILE_EXTENSION.length) == FILE_EXTENSION)
            file.substr(0, file.length - FILE_EXTENSION.length)
        else
            file;
}
private function savePage(content : String)
{
    ensureDirectoryExists(getPageDirectory());
    var out = File.write(getPageFile(), true);
    out.write(content);
    out.close();
}
private function removePage()
{
    FileSystem.deleteFile(getPageFile());
    if(uri != ROOT_PAGE)
        removeEmptyDirectories(getPageDirectory(), dir);
}
private static function ensureDirectoryExists(dir : String)
{
    var base = if(dir.substr(0, 2) == '//')
        '//'
    else
        dir.substr(0, dir.indexOf('\')+1);
Chapter 9: Building Websites with HaXe

```
var path = dir.substr(base.length);
var parts = (~/\[\///\]/g).split(path);
for(part in parts)
{
    base += '/' + part;
    if(!FileSystem.exists(base))
        FileSystem.createDirectory(base);
}

private static function removeEmptyDirectories(dir : String, root : String)
{
    var d = dir;
    while(d != root & FileSystem.exists(d) &
        FileSystem.readDirectory(d).length == 0)
    {
        FileSystem.deleteDirectory(d);
        d = d.substr(0, d.lastIndexOf('/'));
    }
}

enum RequestAction
{
    Edit;
    View;
    Save(content: String);
}
```

To maintain the example short some security good practices have been omitted, but be very careful about malformed URIs. For example, depending on how you handle them, a URI might contain the sequence %00 which will prevent further appending of an extension, or ../ that will navigate potentially through the filesystem and outside the public visible web repository. A good way to prevent malicious intentions is to match the URI correctness using regular expressions that are described in detail in Chapter 8.

The src directory also contains the Page.hx file that includes the definition for the abstract class Page, the PageView.hx, and the PageEdit.hx files contain homonymous class definitions. The hard work is done in the Page class while the other two just add some specialization for the viewing and editing context. The page constructor contains an array of links that will be displayed in the header section of the page. The existing value can be modified at will and new ones can be added to fit the site needs.

The render method just assembles the output from other renderX methods. Those methods have been left separated so that the derived classes may override them individually, avoiding code duplication.

```
class Page
{
    private static var WIKI_HOME_PAGE = 'Wiki - Home Page';
    private static var LOGO_PATH = '/assets/logo.png';
    private static var LOGO_ALT = 'logo wiki';
    private static var BREADCRUMBS_TEXT = 'Where am I?';
    var controller : WikiController;
    var altcontent : String;
    var mainlinks : Array<LinkItem>;
    private function new(controller : WikiController, altcontent : String)
        (continued)
```
Part II: Server Side, JavaScript, and Flash; Oh My!

(continued)

```java
[  
    this.controller = controller;
    this.altcontent = altcontent;
    mainlinks = new Array();
    mainlinks.push(
        {title : "title", uri : "uri"}
    );
}
public function render() : String
{
    return renderHeader() + renderContent() + renderFooter();
}
private function renderHeader() : String
{
    var b = new StringBuf();
    b.add('<!--DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"->
    b.add('"http://www.w3.org/TR/html4/strict.dtd">\n');
    b.add('<!--\n    b.add('<!--title" + getTitle() + '"</title><\n');
    b.add('<!--link href="/assets/main.css" type="text/css" rel="stylesheet" />
');
    b.add('<!--\n    b.add('<!--<body><\n');
    b.add('<!--<div id="header">\n');
    b.add('<!--<div id="wiki-header"> <a href="/" title="" + WIKI_HOME_PAGE + '" titre="">
');
    b.add('<!--<img src="/" alt="/" / >
');
    b.add('<!--</a></div>
');
    b.add('<!--</div>
');
    b.add('<!--</div>
');
    b.add('<!-- mower=\n');
    b.add('<!--<div id="main">\n');
    b.add('<!--<div id="content">\n');
    return b.toString();
}
private function renderContent() : String
{
    return controller.getPageContent(altcontent);
}
private function renderFooter() : String
{
    var b = new StringBuf();
    b.add('<!--\n    b.add('<!--<div id="content">\n');
    b.add('<!--</div>\n');
    b.add('<!--</div>\n');
    b.add('<!--</div>\n');
    b.add('<!--</body>\n');
    b.add('<!--</html>\n');
    return b.toString();
}
private function renderBreadCrumbs() : String
{
    var b : StringBuf = new StringBuf();
    b.add('<!--
    b.add('<!--<ul><\n');
    b.add('<!--</ul>\n');
    b.add('<!--
    b.add('<!--<ul><\n');
    b.add('<!--</ul>\n');
    b.add('<!--
    b.add('<!--<ul><\n');
    b.add('<!--</ul>\n');
    b.add('<!--
    b.add('<!--<ul><\n');
    b.add('<!--</ul>\n');
    return b.toString();
}
```
var list = controller.getBreadcrumbLinks();
for(i in 0 ... list.length)
{
    if(i == list.length -1)
        b.add('         < li > ' + list[i].title + " /n");
    else if(list[i].uri == null)
        b.add('         < li > ' + list[i].title + " » /n");
    else
        b.add('         < li >  < a href="" + list[i].uri + '"' +
            list[i].title + "'" +
            "</a>" +"</li>" +"/n");
    b.add('       < /ul > 
');
b.add('     < /div > 
');
return b.toString();
}
private function renderMainLinks() : String
{
    var b = new StringBuf();
b.add('     < div id="main-links" > 
       < ul > 
');
for(item in mainlinks)
    b.add('         < li >  < a href="" + item.uri + '"' +
        item.title + "'" +
        "</a>" +"</li>" +"/n");
    b.add('       < /ul > 
');
b.add('     < /div > 
');
return b.toString();
}
private function getTitle() : String
{
    return controller.getTitle();
}
}
typedef LinkItem = {
    title : String,
    uri : String
}
The `PageView.hx` file just adds a box in the content area of the page that will be displayed when a message is passed in the constructor. This message is used to pass the save confirmation message.
Part II: Server Side, JavaScript, and Flash; Oh My!

(continued)

```
    return result + super.renderContent();
}

private override function renderFooter()
{
    var b = new StringBuf();
    b.add('\n       < /div > \n');
    b.add('       < div id="page-links" > \n');
    b.add('       < a href="' + controller.uri + '?action=edit" > edit < /a > ');
    b.add(super.renderFooter());
    return b.toString();
}
```

The `PageEdit.hx` file adds a wrapping form and some controls to interact with the page content.

class PageEdit extends Page
{
    private static var EDIT_TITLE_PREFIX = 'Edit: ';
    private static var CONTENT_LABEL = 'The page content goes here:';
    public function new(controller : WikiController, altcontent : String)
    {
        super(controller, altcontent);
    }

    private override function getTitle()
    {
        return EDIT_TITLE_PREFIX + super.getTitle();
    }

    private override function renderHeader()
    {
        var b = new StringBuf();
        b.add(super.renderHeader());
        b.add('         < form action="' + controller.uri + '?action=save" ');
        b.add('         < div class="control" > \n');
        b.add('           < label for="content" > ' + CONTENT_LABEL + ' < /label > \n');
        b.add('           < textarea name="content" > \n');
        b.add('         < /div > \n');
        b.add('         < div class="control" > \n');
        b.add('           < input type="button" onclick="window.location="' + controller.uri + '"">Cancel</input> <input type="submit" name="submit" value="Save" /"> \n');
        b.add('         < /div > \n');
        return b.toString();
    }

    private override function renderFooter()
    {
        var b = new StringBuf();
        b.add('         < textarea name="content" > \n');
        b.add('         < /textarea > \n');
        b.add('         < /div > \n');
        b.add('         < div class="control" > \n');
        b.add('           < input type="button" name="cancel" value="Cancel" /">\n');
        b.add('           < input type="submit" name="submit" value="Save" /">\n');
        b.add('         < /div > \n');
        b.add('         < /form > \n');
        b.add(super.renderFooter());
        return b.toString();
    }
}
Chapter 9: Building Websites with HaXe

The whole code for the Wiki application has been written. Now it is time to compile and see the result.

Add the `Wiki.hxml` file in the project directory. The content is very similar to one in the previous example:

```
-cp src
-neko web/index.n
-main WikiApplication
```

This time the main class is `WikiApplication` and the compiled unit will be `index.n`. Note that whenever an HTTP relative to a directory is invoked, the web server (NekoTools Server by default and `mod_neko` if correctly configured) will look for an `index.n` file inside the directory and will execute it. For this reason you can access the `index.n` without specifying the whole filename but just using the following address once the web server is executed pointing to the web directory.

```
http://localhost:2000/
```

Now you can see the content of the homepage, actually a page that says that no content exists for the homepage. Click the Edit button but wait, page “Not Found”? Why that? This is because, as observed earlier, the web server must be instructed so that all the calls are directed to your `index.n` file. To do this, you have to add a new file `.htaccess` in the web directory. The content of the file must be:

```
<FilesMatch "^([^a-zA-Z0-9-]+)\+$">  
  RewriteEngine On
  RewriteRule (.*) /index.n
</FilesMatch>
```

This simple file instructs the web server so that every URI that does not match an existing file is redirected to the `index.n` file. This feature is available in Apache when the module `mod_rewrite` is activated or in the Neko Server when invoked with the `-rewrite` switch. Therefore, you will probably have to stop your Neko Server and restart it this way:

```
> nekotools server -rewrite
```

Now you can refresh the edit page, insert some contents in the provided form and confirm the submission. The page content is now correctly saved and visualized. The visual result is not very good but you can rapidly improve it by adding a style sheet and a small logo image. The references to those files are already in place in the code produced by the `Page` class. Add a directory `assets` in the `web` directory and create an image with the name `logo.png` (you can use your image editor of choice to do that). For the style sheet, you add the file `main.css` with the following content and watch the result.

```css
* { margin: 0; padding: 0; font-size: 9pt; }
img { border: 0; }
div.message { margin: 10px 0; padding: 4px 4px 4px 32px; font-weight: bold; border: 1px solid; }
div.message { background-color: #d5ffaa; border-color: #4a9500; }
#breadcrumbs { border-bottom: 1px dashed #ccc; padding: 0 0 4px; margin: 0 0 16px; }
#breadcrumbs ul { display: inline; }
#breadcrumbs li { display: inline; margin-right: 4px; font-weight: bold; }
#main { padding: 20px; }
```
The Wiki application can be greatly improved by adding features such as:

- History support for document versions
- User authentication to add editing restrictions
- Resolving security issues
- Wiki text syntax

The implementation of those functionalities and others are left to the user as exercise.

Summary

This chapter explained much about web server development and showed you how to produce a web application with Neko. So far, the following topics were covered:

- What the HTTP protocol is and how it works
- What Web servers and web applications are
- How to use HTML syntax and its most fundamental elements
- How to generate dynamic contents using Neko
- What NekoTools are and how to use it in a development environment
- How to configure mod_neko for Apache
- How to write a Wiki application

The next chapter shows how to write templates to completely separate the business logic from its visual representation.
Separating Design Using Templates

Templates are really a great way to create dynamic contents. In this chapter, you will see how to use them and the following will be discussed:

- What templates are and what they are used for
- How to use the `haxe.Template` class
- How to use the placeholders
- How to provide some logic to templates using `if` and `foreach`
- How to use macros for advanced functionalities
- How to use resources to store templates
- When it is better to use Motion-Twin Templo instead of `haxe.Template`
- How to use advanced features of the Templo system

**What Are Templates?**

A template is a text document that contains placeholders that are dynamically replaced at runtime. The values of the placeholders are set by your application and can change to respond to the interaction with the user.

The biggest advantage in using a template system is the separation of concerns between business logic and user interface (UI), delegating to the template all that is visual and interactive and maintaining the core programming outside the UI context.

Templates are created in a way that a web designer with only HTML experience can write them without any problem; of course he or she will have to stick to some rules described later.
haXe is all about portability but there are exceptions to this rule. Templating relies on the usage of the EReg class (regular expression), which is not available on Flash prior to version 9. This means that the template system will not work on Flash 8 or previous versions. All the other platforms are equally supported.

The Template Class

The Template class is part of the haXe package and it is one of the possible approaches to template processing. Its usage is straightforward. First the class must be instantiated; the constructor accepts a single string argument that is the template content. Then the only public method available, execute(), must be invoked to process the template. The placeholders in the template are replaced by the variable values in the object passed as an argument to the execute() method. This object may be anonymous or a class instance.

```
class Main
{
    static function main()
    {
        var t = new haxe.Template("Hello ::name::!");
        var out = t.execute({ name : 'John Doe' });
        trace(out);
    }
}
```

The result is:

```
Main.hx:7: Hello John Doe!
```

Note how the placeholder ::name:: has been replaced by the content of the corresponding field of the data object. The execute method, also accepts an optional extra argument for macro definitions; the use of this argument is discussed in the “Using Macros” section.

<table>
<thead>
<tr>
<th>Template Class Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new (str:String) : Void</td>
<td>The constructor accepts one argument, which is the content of the template.</td>
</tr>
<tr>
<td>execute(context:Dynamic, ?macros:Dynamic) : String</td>
<td>The first mandatory parameter is an object containing the values that must be used in substitution for the placeholders. The second optional parameter is an object containing macros references.</td>
</tr>
</tbody>
</table>
Chapter 10: Separating Design Using Templates

The Template Syntax

Placeholders are blocks of characters in a template that are marked with the :: prefix and suffix. Thus ::expression:: is the placeholder for a variable named expression. The placeholder variable must be an existent field inside the data object passed to the execute method or a valid expression.

Numeric and Calculation Expressions

An expression can contain a constant numeric value such as ::(123)::, where 123 will be the obvious result. More interesting expressions can be created using the basic math operators (+, -, /, *). So in the following example a simple mathematical expression is used:

```haxe
class Main {
    public static function main()
    {
        var t = new haxe.Template("Seconds in a day ::(24*(60*60))::");
        trace(t.execute(Reflect.empty()));
    }
}
// result in: Main.hx:6: Seconds in a day 86400
```

Note that the participants in the mathematical expression must be grouped in pairs using rounded parentheses; otherwise an exception will be thrown.

```
In the previous example, the passed data object is an empty object because no variables were required. It is important to note that when an empty object has to be created, the correct way is to use the Reflect.empty() method, as in the previous example, and not the {} convention, which is the usual one for those who come from ActionScript or JavaScript. The {} is a valid expression in haXe but has a different meaning; it is a code block and it is equivalent to null, because the block does not contain any return value.
```

Comparison Expressions

In the same way as happens for mathematical expressions, a comparison must be enclosed in rounded parentheses. The following comparison operators are allowed: <, >, <=, >=, and ==.

The result of a comparison operation will output the true or false string, or can be used inside a more complex logical expression as described next. A Boolean value can be negated prefixing it with the ! operator. So the comparison ::!(1<2):: will display false. Boolean values can also be combined with the standard AND (&&) and OR (||) operators.

Dot Expressions

A field in the data object must not be necessarily a primitive value; it can also be an object itself. In this case the standard dot operator will be used to specify the full path to the desired value. Using dot syntax is only possible to access variables and not functions.
A simple example follows to illustrate the dot field access.

```haxe
class Main {
    public static function main() {
        var t = new haxe.Template("::name::'s father is ::father.name::");
        trace(t.execute({ name : "John Doe",
                        father : { name : "John Senior"}}));
    }
}
```

// result is: Main.hx:6: John Doe's father is John Senior

### Logical Expressions

Sometimes it is useful to put a bit of logic inside the template layer. In this way it is possible to discriminate whether a certain piece of output has to be displayed or not. The classic `if/else/elseif` structures can be applied using the following syntax:

```haxe
::if (condition)::
    text to display when condition is true
::elseif (other-condition)::
    text to display when other-condition is true
::else::
    alternative text
::end::
```

Don’t forget to close the logical block with the `::end::` expression. The `::else::` expression and the relative alternative text are optional. More alternatives can be added using the `::elseif condition::` expression.

```haxe
::if (condition)::
    text to display when condition is true
::elseif (other-condition)::
    text to display when other-condition is true
::else::
    alternative text
::end::
```

Rounded parentheses are optional when the condition is expressed using just a variable name but they add clarity to the code.

### Loop Expressions

To loop in a sequence, an object that implements the `Iterable` or `Iterator` type, the `foreach` syntax is used. The syntax is better explained using an example:

```haxe
class Main {
    static function main() {
        var htmlFragment = "<ul>::foreach countries::<li>::name:: (::population::)</li>::end::</ul>";
    }
}```
Chapter 10: Separating Design Using Templates

var list = [
    {
        name: 'United States',
        population: '~298M'
    },
    {
        name: 'Italy',
        population: '~58M'
    },
    {
        name: 'France',
        population: '~63M'
    }
];
var t = new haxe.Template(htmlFragment);
var out = t.execute({ countries: list });
trace(out);

// the result: is Main.hx:20: <ul>
  <li>United States (~298M)</li>
  <li>Italy (~58M)</li>
  <li>France (~63M)</li>
</ul>

In the preceding example, the elements of the data array are objects themselves; inside the ::foreach:: portion of the template the field names of the element objects are used to display values. This is possible because the ::foreach:: structure creates a new context for each element in the loop. It is possible to access to the context value using the special variable name __current__. This is very handy when the elements in the list are not objects but primitives like numbers or strings.

The ::foreach:: block, as the ::if:: block, must be closed using the ::end:: expression or an error will be thrown.

The following example just uses a simple function to return an Iterator object that outputs the numbers in the Fibonacci sequence up to the specified value as argument. Numbers whose values are the sum of the two previous ones compose the Fibonacci sequence. In the template, the __current__ special variable must be used to access the numeric values.

class Main
{
    static function main()
    {
        var template = "::foreach numbers:::__current__::::end::";
        var t = new haxe.Template(template);
        var out = t.execute({ numbers: fibonacci(300) });
        trace(out);
    };
}

static public function fibonacci(upto : Int)
{
    var a = 0;
    var b = 1;
    return {
        hasNext : function() {
            return b <= upto;
        },
        next : function() {
            var v = b;
            var t = new haxe.Template(htmlFragment);
            var out = t.execute({ countries: list });
            trace(out);

            // the result: is Main.hx:20: <ul>
            12/27/07 9:59:21 AM

            c10.indd 255
Part II: Server Side, JavaScript, and Flash; Oh My!

(continued)

```plaintext
b = b + a;
a = v;
return v;
}
}

Importing Templates into Templates

A command to import a template into another template does not exist, but the same effect can be easily obtained executing the first and passing the result into the container template:

```plaintext
class Main {
    static function main()
    {
        var contentTpl = new haxe.Template("the content is: ::content::");
        var content = contentTpl.execute({ content: "content goes here" });
        var containerTpl = new haxe.Template(" <h1> ::title:: </h1> ::content::" );
        var out = containerTpl.execute({ title: "My Page" , content: content });
        trace(out);
    }
}
```

Using Macros

Sometimes you will feel the need to apply more complex transformations to your data before sending them to the screen. Of course, this is the case of transformations needed just for presentation purposes; otherwise, it will be more appropriate to write that code in the business logic of your application.

Macros are just function references or callbacks to your haXe code. Therefore, to have a function you have written in haXe available in a template, you must pass an explicit reference to it using macros. Not having all the functions defined available in the template is a choice of performance and security.

The base syntax to call a macro inside a template is the following:

```plaintext
class Main {
    static function main()
    {
        var t = new haxe.Template("$$sayHelloTo(::user::)" );
        var out = t.execute({
            user: "John"
        },{ 
            sayHelloTo: function(resolve: String->Dynamic, name: String) {
                return "Hello " + name + "!";
            }
        });
        trace(out);
    }
}
```
Chapter 10: Separating Design Using Templates

A new argument has been passed to the `execute()` method. This object contains one or more references to the functions that are needed in the template execution. In the previous example, the `sayHelloTo()` function must be invoked by using the field name prefixed with two dollar characters $$$. The function is invoked by using the user variable as a parameter. In macros it is also possible to pass constant string values; simply put the value without the enclosing :: symbols.

The second thing to note is that the callback function has a formal signature: The first argument must be a function (automatically passed by the template system) that helps to recall other values contained in the template context; other arguments are optional and it is up to the developer to decide what he may need.

This is an example on how to take profit from the passed `resolve()` function.

```haxe
class Main {
    static function main() {
        var t = new haxe.Template("I am $$yearsOld() years old");
        var out = t.execute( {
            birthYear : 1972
        }, {
            yearsOld : function( resolve : String -> Dynamic) {
                // a little bit approximative
                return Date.now().getFullYear() - resolve("birthYear");
            }
        });
        trace(out);
    }
}
```

The resolve function works in the current template context to look for a suitable value.

### Using Resources

Until now, all the template contents have been written inline in the code context. This is not really a good practice but it is useful when templates are used in a test phase or if they are very short. The ideal way to deal with the content of a template is to put it in a resource file, which will be embedded in the result file at compilation time.

A template resource is no more than a plain text file. A file `page.tpl` has been created with the following content:

```html
<html>
<head>
    <title>::title::</title>
</head>
<body>
    ::content::
</body>
</html>
```
Part II: Server Side, JavaScript, and Flash; Oh My!

Now it is necessary to say to the compiler that this resource must be embedded in the output. To do that, the following switch is added to the compilation command or to the .hxml file if one is used:

```
-resource page.tpl@webpage
```

The filename of the resource file is followed by the @ symbol, followed by the name that is used in the code to reference the resource.

Finally, the resource is used in code just using the following syntax:

```
var t = new haxe.Template(Std.resource("webpage"));
```

This way the template content is easily separated from the code that makes use of it.

**When to Use Code in Templates?**

Templates seen so far are very useful in many situations but they have quite an important drawback: They are parsed and interpreted on each call. That is usually not important when the template replacement occurs on the client side (Flash and JavaScript) or the size is not that big, but may be more than an issue if the templates are elaborated on the server side and quite complex and the server has to manage many requests at the same time. In that case a very simple solution is to switch to Templo of the Motion-Twin library, which has a similar syntax but uses compiled templates instead of interpreted ones, which obviously have a very different level of performances.

Performances are not the only concern and for many developers they may not be an issue at all; in fact, Templo is a more complete, feature-rich and mature solution and it works only on Neko. Templo has been designed specifically for (X)HTML and XML. There is an obvious reason behind that decision, considering that most of the development made with haXe is web oriented and that web pages in dynamic websites are almost always produced using templates (see ASP.NET, JSP, PHP, and so on). Nevertheless there are some implications using such an approach, the most important being some of the code used in the Templo system will only work if it is placed in the correct markup context.

Use the templates whenever you have complex formatting but do not abuse them, many times string concatenation will just fit nicely.

**Templo for Server-Side Templates**

As already mentioned the haxelib contains a project named mtwin that is a collection of tools to speed up the development of websites with Neko. One of these tools is Templo, a very complete and efficient template engine. The main qualities of Templo are:

- Template contents are stored in files with the extension .mtt. A template file is just a plain text file. The text format can be a full (X)HTML document or a fragment of it.
- Templates are compiled in Neko files on the first access and are not to be parsed and interpreted on each subsequent call; the Templo system monitors for changes in the source templates and automatically compiles the changed ones when it is necessary. This behavior can be disabled in a production environment to prevent accidental alterations and optimize performances skipping the date check.
Chapter 10: Separating Design Using Templates

- Templates can include other templates.
- Templates have a richer syntax explained in detail next.

**Installing Templo**

Before explaining how to use Templo and the differences with the standard haxe.Template system, the installation process is illustrated. The installation of the library itself is quite simple and the following command from the command prompt/console will do the work:

```
> haxelib install mtwin
```

The last version of the Motion-Twin library is automatically downloaded and installed in the haXe installation directory. At the moment of writing the last version available is 1.2.1 and Templo is installed in the subdirectory of the haXe installation `lib/mtwin/1,2,1/mtwin/templo`

A second optional step must be accomplished to continue the installation. To compile its templates, Templo may need to use the `temploc` executable. This executable can be generated just executing the following command from the Templo installation directory:

```
> haxe temploc.hxml
```

Now that the executable has been generated, it must be accessible so that Templo can use it to compile its templates. In Windows you can make the executable accessible just adding the Templo installation directory to the PATH environment variable. To do this, open the Windows Control Panel, double-click the System icon and in the opened window click the Advanced tab; click the Environment Variables button and in the second list look for a line that starts with Path; select it and click the Edit button. At the end of the second text control, add a semicolon symbol (;) followed by the full path to the Templo installation directory. To conclude, click OK on each of the opened windows. Also the Neko executable must be accessible, but this is normally done during the haXe/Neko installation.

In Linux/Mac OS X you can copy or symlink the executable in an accessible folder (`BIN_PATH`).

The preceding step is not necessary if you do not mean to use the `temploc` executable to manually compile your templates or if you do not want to use the `Loader` class to execute them.

**Using Templo**

The Templo system is composed by a bunch of classes but the one that is really needed to load and execute the templates is the `Template` class. The class constructor accepts the template filename as an argument and its `execute()` method works as in the haxe.Template system accepting a data object, also known as `context`, that is used as a feed to replace the template expressions. Before instantiating the `Template` class, the following static variables from the `Loader` class, must be set properly. As its name implies, the `Loader` class has the responsibility of loading and managing the template files.

The `BASE_DIR` variable is used to set the root location of the templates. All the templates must be included in this directory and will be reachable using their relative path to the base directory as the `Loader` constructor argument.

```
Loader.BASE_DIR = "d:\\templates\\";
```
Part II: Server Side, JavaScript, and Flash; Oh My!

The `TMP_DIR` variable is used to store the compiled templates. If the templates are compiled at run time, you must be sure that the directory has the correct writing permissions or an exception will be thrown.

```xml
Loader.TMP_DIR = "d:\\compiled\\";
```

Macros work a little differently from what you have already seen for the Template system and you can set the `MACROS` static variable to null in order to not use them at all. Later in this chapter they will be explained in detail.

```xml
Loader.MACROS = null;
```

If the `OPTIMIZED` variable is set to false, the templates are compiled on the very first request and every time they are modified thereafter; otherwise the system will assume that the template has already been compiled and is ready to be used and no check will be performed. This setting is usually set to true once all the templates have been successfully compiled and the application is moved to a production environment.

```xml
Loader.OPTIMIZED = false;
```

The default extension for templates is `.mtt`, but it is also possible to use the `.html` and `.tpl` extension. Any other template extension will work too, but if this is the case, some of the (X)HTML-specific features will be disabled. So to resume what has been illustrated so far in a complete example, the following code is put inside the file `template.mtt` in the `templates` directory.

```xml
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>::name:: Profile</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
<h1>::name::</h1>
<ul>
 ::foreach item emails::
   <li>
    ::item.use::
   </li>
 ::end::
</ul>
</body>
</html>
```

Then the `Main.hx` is created with the following content:

```xml
import mtwin.templo.Loader;
import mtwin.templo.Template;
class Main
{
    public static function main()
    {
        Loader.BASE_DIR = "d:\\templates\\"; // replace with a valid dir
        Loader.TMP_DIR = "d:\\compiled\\"; // replace with a valid dir
    }
```

260
Chapter 10: Separating Design Using Templates

Loader.MACROS = null;
Loader.OPTIMIZED = false;
var t = new Template("template.mtt");
var r = t.execute({
    name: "John Doe",
    emails: [{
        email: "john.doe@example.com",
        use: "work"
    }, {
        email: "john@example.com",
        use: "personal"
    }]
});
neko.Lib.print(r);
}
}

Running the previous example in the Nekotools Server or Mod_Neko Apache will output a very basic XHTML page.

Instead of using the mtwin.templo.Template constructor, it is possible to obtain the same results using the mtwin.tempo.Loader constructor. The difference between the two is that the latter uses temploc to compile the templates and, thus, this must be accessible by the system during the execution as described earlier in the “Installing Templo” section.

Remember that to compile the previous example and every piece of code that uses Templo, the Motion-Twin library must be appended to the compilation command line using the switch -lib mtwin.

Differences in Expressions Between haxe.Template and mtwin.Templo

Templo is much smarter than the Template class in many ways because it uses the Neko engine for parsing and generating results. The main differences between the two are:

- Constant values are not limited to integers and can also be float or string values. A string value must be enclosed in single or double quotes.
- Values must not be grouped in pairs; the standard precedence conventions for mathematical operators will be applied.
- The modulo operator is also supported (for example, ::2%3::).
- The dot syntax can also be used to access object functions and not just variables. Variables with getter and setter methods cannot be used; the corresponding function must be used in their place.
- It is possible to access array elements using the square bracket syntax.
- Loop expressions are different in the two implementations and explained in detail later in this chapter.
- All the expression results are filtered and (X)HTML reserved characters are automatically converted in their corresponding entities (for example, the < character is converted to &lt;).
As already discussed, Templo has been seen to work expressly with (X)HTML syntax; this means that certain expressions are only available and useful when associated to tags. The `attr` expression is one of these cases and, as its name suggests, it is used to add an optional attribute in an element.

```html
<a ::attr href 'mailto:'+item.email::<::item.email::</a>
```

The `attr` expression is followed by a keyword that is used as the attribute name, in the preceding example `href`, and by another expression that is used as the attribute value, in the example a string concatenation between a constant value and the `email` field of the `item` object. More than one attribute can be defined in a single `attr` expression; each additional attribute name and value pair must be separated from the previous one by using a semicolon `;` symbol.

When used in an XHTML context, the `attr` expression is smart enough to transform a Boolean value in its corresponding attribute format; as in the following expression:

```html
<input type="checkbox" name="checkme" ::attr checked isChecked:: />
```

It will be transformed in the following if the `isChecked` variable is set to `true`.

```html
<input checked="checked" name="checkme" type="checkbox"/>
```

If the variable value is false the attribute is omitted entirely.

```html
<input name="checkme" type="checkbox"/>
```

That is particularly important because the output is maintained clean with very little effort.

### Raw Expressions

As previously introduced, all the special characters in an expression result are automatically converted in their corresponding entities. The raw expression just prevents this behavior by displaying the content of an unchanged variable. This is handy when the variable contains (X)HTML code fragments.

The following example shows the differences in output using and not using the `raw` expression.

```javascript
import mtwin.templo.Loader;
import mtwin.templo.Template;
var Main =
{
    public static function main()
    {
        Loader.BASE_DIR = "d:\templates\"; // replace with a valid dir
        Loader.TMP_DIR = "d:\compiled\"; // replace with a valid dir
        Loader.MACROS = null;
        Loader.OPTIMIZED = false;
    }
```
Chapter 10: Separating Design Using Templates

```haxe
var t = new Template("raw-template.mtt");
var r = t.execute({ block: 
  " < div > John  &  Jane < /div > " });
neko.Lib.print(r);
```

The content of the template file `raw-template.mtt` is:

```
::raw block::
::block::
```

Finally the output for executing the preceding code is:

```
<div>John &amp; Jane</div>
&lt;div&gt;John &amp; Jane&lt;/div&gt;
```

Having the output of a variable automatically encoded is very useful to prevent all kinds of HTML injections that can represent a very big security vulnerability. The use of `raw` is left for advanced uses only and using it explicitly prevents unwilling misuses.

Logical Expressions

The `if/elseif/else` expressions are exactly the same as described for `haxe.Template`. Templo adds the `cond` expression that is used inside an (X)HTML and permits to toggle its visibility based on a Boolean condition; the element will be visible just in case the condition evaluates to true.

The following template will display the link to an e-mail address if the e-mail value is passed in the data object or the `no email` message otherwise.

```
<a ::cond (email != null  &  &  email != ''):: href="mailto:::email::">::email::</a>
<span ::cond (email == null || email == ''):: >no email</span>
```

Loop Expressions

The syntax for the `foreach` structure in Templo is slightly different from what you have already seen for `haxe.Template`. In Templo it is necessary to label the current element with a name. That name will be used inside the `foreach` context to access the object value and fields.

In the following template, a list of person objects is iterated. Each loop creates a scope variable `person`. Inside the `foreach` structure the person object is accessible as usual using the conventional dot syntax.

```
<ul>
  ::foreach person persons::
  <li>::person.name:: is ::person.age:: years old</li>
::end::
</ul>
```
Part II: Server Side, JavaScript, and Flash; Oh My!

The `foreach` structure also automatically creates a few variables that are commonly used in templates and iterations. Those variables are associated to an object that is accessible using the following naming convention: `repeat.iteration-name.variable-name`, where `iteration-name` is the name defined in the `foreach` expression and `variable-name` is one of the following possibilities:

- **index**: returns the numeric position in the sequence starting from 0.
- **number**: returns the numeric position in the sequence starting from 1.
- **odd**: returns true if index is odd.
- **even**: returns true if index is even.
- **first**: returns true if the current element is the first in the series.
- **last**: returns true if the current element is the last in the series; works only if the sequence has a valid size (different from `null`).
- **size**: returns the number of elements in the series. This variable is available only if the `Iterator` object has one of the methods `get_length()` or `size()`.

The previous example can be slightly changed to show the use of the iteration context. The list element will display something like **John is 35 years old [1 of 10]**. The persons sequence in that case could be the result of a SPOD query.

```html
<ul>
  ::foreach p persons::
  <li>::p.name:: is ::p.age:: old
    [::repeat.p.number:: of ::repeat.p.size::] <li>
  ::end::
</ul>
```

An array in Templo is also accessible using the square brackets syntax. That means that it is possible to return the name of the first person in the list described in the previous example just using the expression `persons[0].name` in the template. It is obvious that this use will not create any iteration context.

The Templo engine provides an alternative way to create loops, the `repeat` expression; it works just like a `foreach` expression but it is applied to an (X)HTML element that is repeated on each cycle. The first `foreach` example can be rewritten in the following way:

```html
<ul>
  <li ::repeat p persons::>
    ::p.name:: is ::p.age:: years old
  </li>
</ul>
```

**set, fill, and use Expressions**

The `set` expression permits creating variables inside a template. Using template variables can be handy in many situations; it is possible to accumulate the values from a list in a sum, or to calculate a value once and print it several times on the same page.
Chapter 10: Separating Design Using Templates

The `set` syntax is as follows:

```plaintext
::set varname = varvalue::
```

Where `varname` is an arbitrary name used to store `varvalue` that is a standard Templo expression. Note that the `set` expression does not print anything; it is necessary to use a standard print expression to display the value of the newly created variable. To sum the values from a cycle it is possible to write a template like this:

```plaintext
::set sum = 0::
::foreach num numbers::
  ::set sum += num::
::end::
The sum total is: ::sum::
```

The `fill` expression works in a very similar way, but instead of capturing the value from an expression, it will capture a fragment of template. In the following example, the template portion between the `fill` expression and the `end` expression is executed and stored in a variable named fragment. This fragment variable can then be reused many times. Remember that if the captured portion contains (X)HTML elements or entities, you will probably want to use the raw expression to display the content unchanged.

```plaintext
::fill fragment::
  <div>a content to repeat: ::content::</div>
::end::
<!-- more code goes here -->
::raw fragment::
```

Template fragments can be shared in Templo using the `use` expression. With this expression, it is possible to include a template inside another. So, you have a template file `login.mtt` that contains just a fragment of (X)HTML relative to a login form; this can be used inside another template in this way:

```plaintext
<div id="context-bar">
  ::use 'login.mtt'::
::end::
</div>
```

Because the template name is an expression, in the previous example a constant string value, it is possible to dynamically change which template to include based on the data passed to the `execute()` method.

Note that the `use` expression must be closed with the `end` expression. This is necessary because of a very nice feature of the `use` expression: The current template can include another template, but it can also be wrapped by a template that acts as a container. Imagine that you have a standard layout and that the only thing that changes is the main portion of the page; you create at least two templates, one for the layout and one for the main section. In the code you are required to execute the main section template, and this will load and be wrapped by the layout template.
Part II: Server Side, JavaScript, and Flash; Oh My!

The following example shows just that. First, a common layout template is realized and saved in the file layout.mtt.

```xml
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>::article.title::</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>
<body>
::raw __content__::
</body>
</html>
```

Now the main template is saved in a file main.mtt.

```xml
::use 'layout.mtt'::
<div class="article-content">
<h1>::article.title::</h1>
<p class="abstract">::article.abstract::</p>
::raw __content__::
</div>
::end::
```

The purpose of the preceding code is to capture the content of the main template in the special variable with predefined name __content__, and then to output its value in the layout template. Note that because the current template and the included wrapper template share the same context, it is possible to refer to the article.title variable also inside the layout template as it is done in the <title> element.

**Using Macros with Templo**

A macro is another way to avoid repeating common pieces of template all over the place. A macro has the aspect of a function in the sense that it accepts arguments that define a scope inside the macro body, but its definition is not HaXe code but template code. During the pre-processing phase of the template execution, the macros are embedded in the process and replace the expressions they are invoked with.

To use macros it is first necessary to create a special-purpose XML file. This file will contain all of the macro definitions and it is structured, as shown in the following schema:

```xml
<macros>
  <macro name="macroName(argument1, argument2)">
    <!-- template fragment goes here -->
  </macro>
</macros>
```

Any macro has a name, macroName in the example, and zero or more arguments. The macros element can contain an unlimited number of macro elements. In a macro body the accessible variables are just the ones passed as arguments.

266
Chapter 10: Separating Design Using Templates

To invoke a macro inside a template the following syntax is used:

```
$macroName(::var1::, other-argument)
```

Where `macroName` is obviously the macro to use, `::var1::` is a context variable and `other-argument` is a constant string. Because the pre-processor just makes a replacement of the macro calls with the corresponding macro bodies, arguments do not require any special treatment like quotations. The exception to this is when a string constant contains rounded parentheses; in this situation, to avoid conflicts, the constant value must be included in curly brackets.

In haXe code, before instantiating the `Template/Loader` class, it will be necessary to set the `Loader.MACROS` static variable to point to the macros XML file. In the example, the macro file has the name `macros.xml` and it contains the following code:

```
<macros>
  <macro name="image(uri, desc, link)">
    ::if (link == '' || link == null)::
      <img ::attr src uri; alt desc:: />
    ::else::
      <a ::attr href link:: > <img ::attr src uri; alt desc:: /></a>
    ::end::
  </macro>
  <macro name="zebraList(list)">
    <ul>
      ::foreach item list::
        ::if isOdd::
          ::set class = 'odd-row'::
        ::else::
          ::set class = 'even-row'::
        ::end::
        <li ::attr class class :: > ::item:: 
        ::end::
      </li>
    </ul>
  </macro>
</macros>
```

Thus, two macros are defined: The first creates the tag for an image element and accepts a URI for the image, and a description; it optionally accepts a third URI argument if the image has to be used as a hyperlink. The second macro accepts a list (an `Iterator` or `Iterable` object) and displays its elements in a standard (X)HTML list.

The template body that makes use of the preceding macros, can be something like this:

```
$image(::image.uri::, ::image.alt::, ::image.link::)
$zebraList(::names::)
```

The macros can remove a lot of redundancy in the template code; errors are easier to manage because they have to be fixed just once in the macros file, the template code is easier to read because it is shorter and because macro names add semantic to it.
Part II: Server Side, JavaScript, and Flash; Oh My!

On the haXe side, the code to make the example work is the following.

```javascript
import mtwin.templo.Loader;
import mtwin.templo.Template;
class Main {
    public static function main()
    {
        Loader.BASE_DIR = "d:\\templates\\"; // replace with a valid dir
        Loader.TMP_DIR = "d:\\compiled\\"; // replace with a valid dir
        Loader.MACROS = "macros.xml";
        Loader.OPTIMIZED = false;
        var t = new Template("macros-template.mtt");
        var r = t.execute({
            image: {
                uri : 'images/logo.png',
                alt : 'image description',
                link: 'http://www.example.com/
            },
            names: [
                "George Washington",
                "John Adams",
                "Thomas Jefferson",
                "James Madison",
                "James Monroe",
                "John Quincy Adams"
            ]
        });
        neko.Lib.print(r);
    }
}
```

### Compiling Templates Manually

As already discussed, when the `Loader.OPTIMIZED` static variable is set to true, the Templo system will not automatically compile the templates and it will expect that the compiled are already available in the `Loader.TMP_DIR`. To compile the template manually, you need to use the `temploc` executable that accepts the parameters described in Table 10-2.

#### Table 10-2

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o /destination/directory</td>
<td>Specifies the directory in which the template files will be compiled.</td>
</tr>
<tr>
<td>-m macrofile.xml</td>
<td>Specifies the name of the file that contains the macro definitions. The macro file must reside in the same folder or a subfolder of the templates.</td>
</tr>
<tr>
<td>-r /templates/directory</td>
<td>Specifies the directory containing the templates.</td>
</tr>
<tr>
<td>filename[filename]+</td>
<td>Specifies one or more space-separated names of template files.</td>
</tr>
</tbody>
</table>
Chapter 10: Separating Design Using Templates

Summary

As you can see, using templates is not that hard and they can really make your day saving a lot of awful string concatenation and cleaning up a lot of messy code. In this chapter, the following topics were covered:

- What a template system is
- How it is implemented in haXe
- How to use placeholders and add some logic to templates
- How to use macros and callbacks
- What resources are used to store templates
- How to replace haxe.Template with Templo

The next chapter illustrates some of the magic of programming with haXe and Neko on the server side and how to get the best results.
Performing Server-Side Trickery

We have a cumulative experience of over twenty years developing with JavaScript and Flash ActionScript, and many of those years have been spent developing with the commercial tools provided by Macromedia (now Adobe). Since discovering the MTASC compiler for ActionScript development, and now the haXe compiler, we have marveled and pondered at just how much time we have spent dealing with these two languages within the confines of the development tools we had to handle, and wonder how much more could have been achieved had haXe been around during our entire development career. That is not to say that the tools we had were poor. Far from it. As they say, a good workman never blames his tools. Only, there have always been issues with JavaScript and ActionScript development that we can now say are almost entirely diminished thanks to the haXe compiler. Despite the sheer power and capabilities of haXe that we have come to love, though, it is not haXe that we consider as the be-all-and-end-all of development tools. That we save for the Neko Virtual Machine.

In this chapter, you’ll begin to look at the Neko Virtual Machine in more depth, including:

- How to use its standard libraries
- How to access and manipulate databases
- How to use the SPOD system to access and manipulate databases more easily
- How to access and manipulate the filesystem
- How to extend your Exceptions class to incorporate file-based logging

Introducing Neko

Neko is a collection of tools, including a Neko-specific compiler, the Neko virtual machine, and a few tools to aid development. The Neko language itself is a dynamic procedural style language oriented toward code generators and the like, with little use for handwriting code yourself.
Part II: Server Side, JavaScript, and Flash: Oh My!

The compiler also supports two other higher-level variants of the Neko language: NekoML and NXML. The former of these two languages is a functional style language with strong pattern matching capabilities, ideal for compiler creation, while the latter of the two is written in an XML format and is mainly targeted as an easy way for applications to output Neko compilable code.

The Neko languages will not be covered in any depth as they fall outside of the scope of this book. At present, little has been documented about their usage, thus haXe should be considered as the primary language and compiler for use with the Neko platform. Besides, the Neko virtual machine is the gem in the toolset and will therefore necessitate all further focus.

The Neko Virtual Machine

The Neko virtual machine is without a doubt the most incredible tool we have ever laid our hands on. It is light, fast, and so easy to develop against that one can’t help running experiments with it to see just what it is capable of. There is simply no end to the possibilities it provides, and only a little exploring is all it takes to get you hooked.

Written in C, it is fully capable of compiling and running on numerous operating systems, including Mac OSX, Linux, Windows, and possibly more in the future. Its C foreign function interface is incredibly simple, making even Ruby seem a complex virtual machine to extend. Plus, the Neko virtual machine requires an absolute minimal installation, meaning desktop applications written for the Neko platform are very easy to get running on end-user machines.

The Neko virtual machine is a dynamically linked library (.dylib, .so, or .dll depending on the operating system you are using). Its task is to convert compiled Neko bytecode, normally contained in files with an .n extension, into executable code. However, in order to do this, you need to provide access to the containing file. This can be done by executing the neko executable contained in the Neko distribution and passing it the location of the file to execute, or by linking the Neko dynamic library into an executable application of your own and passing the file location details using code.

As you can imagine, all of this flexibility means there are a great many ways to do a great many things, so you’ll concentrate on much of the basics in this chapter and extend upon this as you progress through the book.

The Neko Standard Libraries

The Neko standard libraries can be split into two parts: the .ndll (Neko dynamic link library) files containing the compiled C language code and a series of haXe classes that either wrap the .ndll functionality for use in your own haXe applications (provided they’re compiled for the Neko virtual machine) or provide classes that extend these wrappers. This kind of library partnership can be seen in other languages, such as Python, PHP, and Microsoft’s Active Server Pages. Though, we guarantee you’ll never again see the pure simplicity and power capable within the relationship between Neko and its libraries.

As with other languages, the Neko libraries exist to provide various types of functionality, such as the ability to access and manipulate databases and the filesystem. These libraries can also help with the more mundane tasks that are otherwise absent from the main virtual machine, and which some virtual machines for other platforms come fully equipped. By separating the various types of functionality from the virtual machine, you are only obliged to include the libraries that your application requires, thus keeping file sizes for distributed applications to a minimum.
Chapter 11: Performing Server-Side Trickery

You’ve already covered several of the Neko standard libraries, though you might not realize it. For instance, XML and regular expression support are both provided by the regular expression C code found in `regexp.ndll`. Now, although the Neko Virtual Machine requires that the `regexp.ndll` file be included with your application for it to function, it is easily possible for your application to barely use this library at all. On the other hand, the JavaScript and ActionScript 3 virtual machines come with the regular expression support built in. Of course, not having the choice to include or exclude the `regexp.ndll` makes this example a bit pointless, but you’ll appreciate the point being made.

Working with Databases

One aspect, among many, that we love about Neko is its ability to work with databases in a very simple way. Supporting all manner of queries offered by the databases themselves, the haXe database framework provides more than enough to satisfy all your relational information needs.

Each of the databases currently supported by the Neko standard libraries provide the same array of functions for each, so, aside from the usual T/SQL variations, you can access and modify each database type with no necessary code alterations. This means that any application you create for distribution can literally support numerous database types without the intervention of dark magic (as often performed by one of our database savvy colleagues).

At the time of this writing, the Neko platform only provides support for four types of database API. The MySQL and SQLite libraries come with the Neko distribution, while the PostgreSQL wrapper and ODBC driver, developed by Lee McColl Sylvester, fully supports the Neko/haXe SPOD system and is available for download at www.designrealm.co.uk.

To make life extra easy, the haXe distro comes complete with a very capable persistent database objects layer called the SPOD. If you’ve ever used the hibernate framework for Java and C#, you could consider the SPOD as a very lightweight version of this framework, only with a much shallower learning curve and a much smaller feature list. It is more than powerful enough for the majority of uses, though, and certainly beats a poke in the eye with a sharp stick.

If you are new to persistent object layers for databases, you’re definitely in for a treat. The Neko SPOD system is a clever little invention that allows complete mapping between objects and database tables, providing the means to query and update the contained data without having to write a single line of SQL.

Choosing a Database

Okay, so you’re not exactly spoiled for choice when it comes to selecting a database for use in your haXe applications. Quite frankly, though, you couldn’t really ask for more. The three database servers that the Neko Virtual Machine does support natively cover a broad area of capabilities, so requesting the services of an unsupported database can only really be justified in the case of existing data sources.

Choosing a database should be reliant upon the actual needs of the application you’re building. Each database server has its own major pros and perils, but the first hurdle you’ll likely face is with being honest with yourself about the likely traffic and capabilities exacted upon it. For instance, will your website really receive more than 100 concurrent visitors or even as much as 100,000 visitors in one day? Is absolute minimal latency a must? Will you be storing the entire works of William Shakespeare or are you merely storing a handful of blog entries? By determining your actual needs rather than your fantasy
requirements, you ensure that you’re not overextending your development time and capabilities. At the same time, though, you must still provide a little growing room for that just in case factor. You never truly know, your website could become the next YouTube.com.

**The PostgreSQL Database**

PostgreSQL originally started out its life as a project to provide an open source enterprise-level database server while maintaining absolute conformity with officially published standards and specifications. As the database matured and finally reached a version supporting all you would expect from a commercial enterprise system, the product then continued to be optimized in order to make it as fast and efficient as possible. Now it is a world-class system standing in the top three database systems used in Japan and backed by major companies such as Fujitsu.

In its current rendition, the PostgreSQL database could quite easily be considered as an open source competitor to the Oracle enterprise server, providing many advanced features including stored procedures, triggers, and its own scripting language on par with PL/SQL itself.

All these features are not without their cost, though, as although PostgreSQL could quite possibly satisfy any enterprise data storage requirement, it does have a level of complexity and requires careful planning and regular tuning.

**The MySQL Database**

The MySQL database server is probably one of the most popular database servers in the world and has been for sometime. At present, there is little difference between the offering provided by the PostgreSQL database and that of MySQL. However, the development of the MySQL server from its original inception has put ease of use and efficiency at the forefront, while only concentrating wholly on standards conformity and breadth of features during its later renditions.

Another benefit of the MySQL database is that it generally has a much larger acceptance than other open source databases, meaning that distributed web applications will have little trouble finding a host that supports it.

**The SQLite Database**

The SQLite database is a masterpiece. That’s all there is to it. As a believer that nothing is faster than text, we have always preferred storing data in XML files or flat text files than waste valuable server resources using a database server for minimal information. By minimal, we mean less than one gigabyte of data, though typically, much less than that. Of course, storing one gigabyte of data in text files would be unruly, but unless the majority of your data is floating in memory or you have your data particularly well indexed, then simple flat text files will rule out in access times, every time.

The SQLite database was built with this information in mind. In truth, it is little more than a flat text file with a database engine, but it has been built well, and in such a way that text files will never be sufficient again. The philosophy behind SQLite is that 95 percent of websites are small to medium traffic sites with small to medium data requirements. This means that, for most websites, using an enterprise-level database to store the website data is overkill, thus wasting valuable resources, server real-estate and development time.
Chapter 11: Performing Server-Side Trickery

Here are the bottom-line pros for using a SQLite database:

- Requires absolutely no dependencies (excepting libsqlite on Linux).
- Runs on almost any operating system, including embedded systems.
- Database files are independent of drive formatting and CPU, meaning they are fully portable.
- Supports most of the SQL92 specification.
- Suitable for sites receiving 100,000 hits per day, and can support sites with up to 1,000,000 hits per day.
- Suitable for sites with around 200 to 300 concurrently browsing visitors.
- Incredibly fast access times when accessed locally (that is, by using a scripting language on a local SQLite file), even so far as to be faster than most enterprise-level systems.
- Easier to back up. No need to normalize the database before backups are made, plus the database need only be downloaded to fully back up.
- All data inserts, deletes, and updates are transactional, plus any system failure, including a complete crash, will safely roll back a transaction.

And, here are the cons:

- Very slight latency if accessed remotely.
- Dirty pages (uncommitted data) can take around 256KB for every 1MB of stored data.
- Not really suitable for websites exceeding a couple of gigabytes in data.
- As the database is a file, all read and write locks exist database wide rather than object wide, meaning too many simultaneous visitors (300+ on average) slow down the user experience.

Conclusion

As noted above, the SQLite database is suitable for 95 percent of websites. However, if you think a little, SQLite is the perfect companion to the Neko Virtual Machine. Both are small in file size, require no unusual dependencies, run on most platforms, and are easy to develop against. What’s more, as the Neko Virtual Machine is perfect for desktop applications, too, that should surely raise the usability of these two tools, somewhat.

At the end of it all, the conclusion boils down to the original question: Will your application exceed the constraints of the SQLite database? If no, then use it, as it will never let you down (don’t quote us on that). However, if your needs are a little excessive when compared to the usual development brief, then it might be time to pull out the bigger guns and opt for either MySQL or PostgreSQL. Either is sufficient in its current state, and indeed, there is little between them now besides the sheer preference of the developer. Alternatively, help us test our ODBC driver by using it with your favorite database. The choice is yours.
Part II: Server Side, JavaScript, and Flash: Oh My!

As SQLite provides everything you need to get up and running with a database and is fully included in the haXe distro, most of the examples in this chapter will be SQLite oriented. Besides, thanks to the great set of cross-database libraries that come with haXe, it makes little difference which database server you choose to run the examples with.

Connecting to the Database

Connecting to a database is one of the few times that you’ll need to write differing code depending on the target database, simply because most databases require a differing connection string. Once connected, though, it’s pretty much plain sailing as you’ll find almost all other functionality identical.

Connecting to a SQLite Database

Connecting to a database requires a single line of code, regardless of the database you’re using, though the parameter requirements do differ quite considerably. For instance, SQLite, being a file-based database, only requires you to pass a single parameter value for the location of the database file. You do this when calling the static `open` method of the `neko.db.Sqlite` class, which in turn returns an implementation of the `neko.db.Connection` interface required for all future database queries:

```javascript
var conn = neko.db.Sqlite.open( "mydatabase.db3" );
```

If the SQLite database file does not exist when trying to open it, then the file is automatically created. However, if you do not want the database to be automatically created upon failing to locate the file, you can use the Neko filesystem classes to detect the file’s existence before making the call. Filesystem manipulation will be discussed later in this chapter.

Connecting to a PostgreSQL Database

The PostgreSQL connection works in a very similar way to the SQLite connection. The only visible difference, aside from the different database class name, is that instead of accepting the location of the database file, it expects a valid PostgreSQL connection string:

```javascript
var connStr = "dbname=simpledb user=Lee";
var conn = neko.db.Postgresql.open( connStr );
```

You can check if the connection was unsuccessful by comparing the return value with `null`.

Connecting to a MySQL Database

The MySQL database connection also requires a single parameter, though it is an equivalent of passing six. This time, the parameter is passed to the static `connect` method of the `neko.db.Mysql` class:

```javascript
var conn = neko.db.Mysql.connect({
    host : "localhost",
    port : 3306,
    user : "root",
    pass : "",
    socket : null,
    database : "MyBase"
});
```
Chapter 11: Performing Server-Side Trickery

The parameter passed to `connect` is of an object type and must contain all six properties listed in the following table, as they are necessary to achieve a successful connection. Once connected, the method returns an implementation of the `neko.db.Connection` interface. Table 11-1 details the properties accepted by the MySQL `connect` method.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>host</code> : String</td>
<td>The hostname/IP Address of the server hosting the MySQL database</td>
<td>Hostname or IP</td>
</tr>
<tr>
<td><code>port</code> : Int</td>
<td>The port that the MySQL database is listening</td>
<td>Usually defaults to 3306</td>
</tr>
<tr>
<td><code>user</code> : String</td>
<td>Name of the user to connect</td>
<td></td>
</tr>
<tr>
<td><code>pass</code> : String</td>
<td>Password of the user to connect</td>
<td></td>
</tr>
<tr>
<td><code>socket</code> : String</td>
<td>On UNIX systems, set to the id of the socket to connect to the MySQL database</td>
<td>Set to null for default connection or the id of the socket to connect</td>
</tr>
<tr>
<td><code>database</code> : String</td>
<td>The valid name of the database hosted by the MySQL Server that you wish to connect</td>
<td></td>
</tr>
</tbody>
</table>

**Connecting with ODBC**

The ODBC driver provides a *one option fits all* approach by accepting a single value to its `connect` method. However, the value passed will vary from database to database, so you may need to look this up for your particular flavor of database.

```
var conn = neko.db.ODBC.connect( someMiscellaneousConnString );
```

**The Connection Class**

Once a connection has been made to your database, you will then perform most of your regular database interaction using an instance of a `Connection` class that implements the `neko.db.Connection` interface. This `Connection` class is where the communication between your haXe application and the Neko database `.ndll` module is performed. It allows you to perform tasks such as running queries, creating, committing, and rolling back transactions and performing simple data conversions such as escaping illegal string characters and quotes. Table 11-2 lists the methods of the `neko.db.Connection` interface.
Part II: Server Side, JavaScript, and Flash: Oh My!

Table 11-2

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>close() : Void</td>
<td>Closes the current connection</td>
</tr>
<tr>
<td>commit() : Void</td>
<td>Commits the current transaction</td>
</tr>
<tr>
<td>dbName() : String</td>
<td>Returns the name of the currently connected database</td>
</tr>
<tr>
<td>escape( s : String ) : String</td>
<td>Escapes illegal characters in the passed String value</td>
</tr>
<tr>
<td>lastInsertId() : Int</td>
<td>Returns the last inserted record id</td>
</tr>
<tr>
<td>quote( s : String ) : String</td>
<td>Performs the escape method on the passed String value, then surrounds it with single quote characters</td>
</tr>
<tr>
<td>request( sql : String ) : neko.db.ResultSet</td>
<td>Executes a given SQL statement and returns a ResultSet for any records returned by the database</td>
</tr>
<tr>
<td>rollback() : Void</td>
<td>Rolls back the current transaction</td>
</tr>
<tr>
<td>startTransaction() : Void</td>
<td>Begins a new transaction for the current connection object</td>
</tr>
</tbody>
</table>

request()

request will be the method you most use in the Connection objects arsenal. It allows you to execute all of your SQL strings against the connected database, whether it returns a value or not. You will need to make sure that the SQL you are passing to the request method is suitable for the database you are connected to, though it is possible to write SQL that is acceptable by either of the supported databases.

Creating a Table

The first time a database is created, you will need to define its structure. Performing this within haXe can be very useful, especially if your application is distributed to end users using the SQLite database. This way, you can create your database when your application is first run, reducing the number of files to be distributed.

Here is an example creating a new database called Wrox.db3 and populating it with a table called Author:

```haxe
class CreateDB
{
    public static function main()
    {
        var dbLoc = 'Wrox.db3';
        var dbFactory = create( dbLoc );
    }

    public static function create( loc : String )
    {
        // Code for creating the database and table
    }
}
```
Chapter 11: Performing Server-Side Trickery

```javascript
{  
  // open the database
  var cnx = neko.db.Sqlite.open( loc );
  // execute the 'create table' sql
  cnx.request( "CREATE TABLE Author (
      int_id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
      var_username TEXT NOT NULL,
      var_password TEXT NOT NULL,
      var_email TEXT NOT NULL,
      dte_added DATETIME NOT NULL )" );
  // close the database connection
  cnx.close();
}
```

You’ll continue to use this database for the rest of the examples in this chapter, so go ahead and run the code. As you can see, the code necessary to create a database is incredibly small when compared to other so-say simple languages such as Visual Basic. In only three lines of code, the database was created and opened, a table was added, and then the connection was closed with all resources released.

Populating the Database

Thanks to the simplicity of the haXe database framework, you can now populate your database with little difference in code to the previous example:

```javascript
class PrePopulateDB
{
  private static var dte : Date = Date.now();
  private static var data : Array<Dynamic> = [{
    user:"Lee",
    pass:"secret",
    email:"lee@designrealm.co.uk",
    added:Date.now().toString()
  },{
    user:"James",
    pass:"pass",
    email:"james@vdev.co.uk",
    added:Date.now().toString()
  },{
    user:"Edwin",
    pass:"letmein",
    email:"edwin@screenweaver.com",
    added:Date.now().toString()
  }];
  public static function main()
  {
    var dbLoc = "Wrox.db3";
    var dbPop = populate( dbLoc );
  }
  public static function populate( loc : String )
  {
    // open the database
    var cnx = neko.db.Sqlite.open( loc );
    var sql : String;
    (continued)
```
As you can see, much of the bulk of this class is taken up by the array containing the data required to populate the database and the loop that constructs the SQL string. The rest of the class is pretty much identical to that of CreateDB.

You can generally use the same connection class functionality when updating and deleting records from the database. For example, to update a record, you could do the following:

```java
public static function update( loc : String )
{
    var cnx = neko.db.Sqlite.open( loc );
    var sql = "UPDATE Author SET " + cnx.quote( "var_password = " + cnx.quote( "h4X3ruL35" ) );
    sql += " WHERE var_username = " + cnx.quote( "Lee" );
    cnx.request( sql );
    cnx.close();
}
```

While deleting a record might look like this:

```java
public static function main()
{
    var dbLoc = "Wrox.db3";
    var dbPop = update( dbLoc );
}
```
Chapter 11: Performing Server-Side Trickery

```javascript
{  
  var dbLoc = "Wrox.db3";
  var dbPop = deleteRec( dbLoc );
}
public static function deleteRec( loc : String )
{
  // open the database
  var cnx = neko.db.Sqlite.open( loc );
  // construct the sql string
  var sql = "DELETE FROM Author ";
  sql += " WHERE var_username = " + cnx.quote( "Lee" );
  // execute the sql
  cnx.request( sql );
  // close the database connection
  cnx.close();
}
```

As you can see from these examples, when passing in data from variables to construct the SQL strings ready for consumption by the request method, the variable data is first passed to the Connection.quote method. This is because, as honorable as your application users are, you do not want to rely on your users formatting the text for acceptance by the database. At best, any rogue quotes within a given string might cause the SQL execution to fail, while at worst, purposeful malicious SQL code may be passed resulting in dire consequences for your data.

dbName() and lastInsertId()

dbName and lastInsertId may as well be getter methods, as they perform no actual calculation and exist only to return a value. The dbName method simply provides a static string depicting the type of database server. This, currently, can be SQLite, MySQL or PostgreSQL. The point of this method is so you can compare it against a literal representation of these values in a conditional expression so as to determine the database type that a Connection instance is associated with.

The lastInsertId method is useful when you wish to retain the record id of the last inserted record. For example, it could be that you want to add a record to a table, but also insert several other records to look up tables that rely on the first records id. To do this, you could simply store the return value from lastInsertId into a variable, and use the variable in each further relative insert.

Care must be taken when using the lastInsertId method with the SQLite database, as all SQLite tables possess a hidden record id column to which the lastInsertId method returns. If you choose to include your own record id column, as carried out in the previous examples, you must facilitate your own id retrieval functionality. If, however, you choose to use the hidden record id column, you can include this column in your own SQL strings by referencing the column as either ROWID, _ROWID_, or OID.

Transactions

Transactions provide a great way to ensure data integrity when inserting, updating, or deleting multiple records. For example, if you need to insert or update two tables at once, and need to guarantee that either both SQL executions are successful or both are unsuccessful, then you can perform the executions within a transaction and commit the transaction only if both are successful; otherwise you roll the transaction back to its previous state.
At the time of this writing, the Connection class provides full transactional capabilities for all current databases accessible by the haXe language. However, as new wrappers materialize, it is recommended that you check their level of transaction support.

As with all the haXe database features, using database transactions in haXe is dead simple. Let’s jump straight in and take a look at an example:

```haXe
class Transaction
{
    public static function main()
    {
        var dbLoc = "Wrox.db3";
        var dbPop = transact( dbLoc );
    }

    public static function transact( loc : String )
    {
        // open the database
        var cnx = neko.db.Sqlite.open( loc );
        // begin the transaction
        cnx.startTransaction();
        // construct the sql string
        var sql = "INSERT INTO Author ( var_username, var_password, var_email, dte_added ) VALUES ( ";
        sql += "'Nicolas', 'mtw1n', 'ncannasse@motion-twin.com', ";
        sql += Date.now().toString() + "' )";
        // execute the sql
        cnx.request( sql );
        // construct the second sql string - should return 0 records
        sql = "SELECT COUNT(*) FROM Author WHERE var_username='MetalMickey'";
        // execute second sql
        var rec = cnx.request( sql );
        // if the returned value is zero, rollback the transaction
        if ( rec.getIntResult( 0 ) == 0 )
            cnx.rollback();
        else
            cnx.commit();
        // close the database connection
        cnx.close();
    }
}
```

As you can see, much of the code isn’t so different from what you’ve already accomplished with previous examples. The main difference is, before any of the SQL code was executed, you started a new transaction for the current Connection object. This transaction runs until commit or rollback is called, or the Connection object is closed, either with the close method or by force, such as an application crash. Any insert, update or delete statement executed after the startTransaction call will be rolled back at any of these points except when commit is called successfully. This provides complete protection for your data integrity.

Another new feature in the aforementioned Transaction class that has not yet been covered involves dealing with data returned from an executed query. You’ll look at this in great detail, next.
Chapter 11: Performing Server-Side Trickery

The ResultSet Class

Each database supported by the haXe language also makes use of a ResultSet class that implements neko.db.ResultSet. It is the primary purpose of this class to act as a container for any data that should return from an executed query and, thus, provide means of access to that data.

Table 11-3 details the methods of the neko.db.ResultSet interface:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length() : Int</td>
<td>Returns the number of records from a select, update, or delete query</td>
</tr>
<tr>
<td>nfields() : Int</td>
<td>Returns the number of columns in the ResultSet</td>
</tr>
<tr>
<td>getFloatResult(val : Int) : Float</td>
<td>Returns a value as type Float from a given column index</td>
</tr>
<tr>
<td>getIntResult(val : Int) : Int</td>
<td>Returns a value as type Int from a given column index</td>
</tr>
<tr>
<td>getResult(val : Int) : String</td>
<td>Returns a value as type String from a given column index</td>
</tr>
<tr>
<td>hasNext() : Bool</td>
<td>Facilitates the Iterator interface</td>
</tr>
<tr>
<td>next() : Dynamic</td>
<td>Facilitates the Iterator interface</td>
</tr>
<tr>
<td>results() : List&lt;Dynamic&gt;</td>
<td>Returns the ResultSet content as a List object</td>
</tr>
</tbody>
</table>

Accessing Fields by Name

When records are returned in a ResultSet object, any values with an associated column name referenced within the returned data is appended to the object as accessible properties. This means that, providing you know the column name or alias specified for a particular field of data, you can access it as though it were a member variable of an instance of a class. For example, supposing you were to execute the following query:

```javascript
var rec = cnx.request( "SELECT * FROM Author ORDER BY var_username DESC LIMIT 1" );
```

You could then access each of the data fields returned from the query execution like this:

```javascript
neko.Lib.print( "User : " + rec.var_username + " | Pass : " + rec.var_password );
```

283
The data appended to the ResultSet in this fashion can also be accessed using reflection or through an iterator in much the same way as other objects, though the ResultSet does provide the necessary methods required to treat the ResultSet itself as an iterator.

**Accessing Fields by Index**

Very often, you will execute queries that do not provide a field name for one or more returned values, such as when an aggregate function like count or sum is used. Now, you could opt to provide an alias for the values returned from these functions, thus facilitating a column name, but haXe provides an alternative option using the getResult, getIntResult, and getFloatResult methods that can be just as effective.

Each of these methods accepts an integer value that represents the index of the item in the returned record, where zero is the first item. Then, depending on the specified method, it will return the value contained in that index. If the value stored at the given location is not of the type returned by the method, then it will attempt to convert the value. This is fine if you want to convert an integer to a float or a string, but can have unexpected results if, say, it converted a string to an integer:

```java
class DBQuery {
    public static function main() {
        var dbLoc = "Wrox.db3";
        var dbPop = query( dbLoc );
    }
    public static function query( loc : String ) {
        // open the connection
        var cnx = neko.db.Sqlite.open( loc );
        // execute the request
        var rec = cnx.request( "SELECT var_username FROM Author" );
        // retrieve data as Float, Int and String
        var f : Float = rec.getFloatResult( 0 );
        var i : Int = rec.getIntResult( 0 );
        var s : String = rec.getResult( 0 );
        // trace the results
        trace( f + ":" + i + "::" + s );
        // close the connection
        cnx.close();
    }
}
```

When compiled and run, this class printed out the values:

```
0:0:James
```

Of course, your output may differ depending on the content in your database.

**The nfields Property**

When accessing data in a ResultSet, you might not be 100 percent sure of the structure of the data you have queried. For example, the SQL string you passed to the request method might be accessing a table where little is known of its content, while at the same time, it could be imperative that the information is rendered in its entirety.
Chapter 11: Performing Server-Side Trickery

For such occasions, the ResultSet provides an nfields property that enables the developer to query the exact number of columns returned by the request. This way, you can choose to iterate through each column while the column index is less than the value of the nfields property, thus guaranteeing access to all of the contained data.

**Iterating Through a ResultSet**

The ResultSet provides the necessary methods as detailed by the typedef Iterator. This means that a populated ResultSet object can be iterated in a loop allowing easy access to its content. This is an invaluable feature when dealing with ResultSet containing more than one record:

```java
class DBIteration {
    public static function main() {
        var dbLoc = "Wrox.db3";
        iter( dbLoc );
    }

    public static function iter( loc : String ) {
        // open the connection
        var cnx = neko.db.Sqlite.open( loc );

        var sql = "SELECT * FROM Author";
        // execute the query
        var rec = cnx.request( sql );
        // loop through the rows and print user data
        for ( user in rec )
            neko.Lib.print( user.var_username + "\t" + user.var_password + "\n" );

        // close the connection
        cnx.close();
    }
}
```

As you can see, looping through each record in the ResultSet proves to be a very easy task indeed. The ResultSet object also provides the means to export the contained data in a List object for easy integration into functions and methods that accept the List object as a parameter.

If you need to access specific records in a ResultSet and would prefer access to the records via an index, then you can quite comfortably convert the ResultSet into an array, as shown in this example:

```java
class DBArray {
    public static function main() {
        var dbLoc = "Wrox.db3";
        arrayFromDB( dbLoc );
    }

    public static function arrayFromDB( loc : String ) {
        // open database connection
        var cnx = neko.db.Sqlite.open( loc );

        // (continued)
    }
}
```
Part II: Server Side, JavaScript, and Flash: Oh My!

(continued)

```javascript
var sql = "SELECT * FROM Author";
// execute query
var rec = cnx.request( sql );
// convert ResultSet to an array
var arr = Lambda.array( rec.results() );
// loop through array while printing values
for (i in 0...arr.length)
{
    neko.Lib.print( i + ": " + arr[i].var_username );
    neko.Lib.print( "\t" + arr[i].var_password + "\n" );
}

// close the connection
cnx.close();
```

This produces much the same result as iterating through the ResultSet. However, the array conversion
does provide the facility to choose individual records at your leisure while not compromising on the
content and structure.

**The Length Property**

The `length` property of the `ResultSet` class is a nifty way to find out how many records were returned
in a select query. While not as useful as it could be, thanks to iterators, the data can still come in handy
on occasion. The `length` property truly comes into its own, however, when used with an update query.
As previously noted, all successful executions of the `request` method return a valid `ResultSet` object,
whether it contains records or not. The difference is, if the query executed by the `request` method is that
of an update query, then the `length` property of the returned `ResultSet` contains the number of records
updated in that request.

**Using the SPOD System**

The SPOD system, or Simple Persistent Objects Database system, is a framework extending the
capabilities of the aforementioned database support. Utilizing only two classes, `neko.db.Object` and
`neko.db.Manager`, the SPOD system increases the simplicity of the haXe and Neko database support,
bridging the gap between object orientation and relational data storage by providing object to relational
mapping. To put this another way, by providing a map between objects and their relative database
tables, you can provide database updates, inserts, and deletes by simply creating or updating the
mapped objects and then calling an associated method.

There are several other systems that do this with other languages. The most notable is the Hibernate
framework for Java and the .NET framework. Now, the haXe SPOD system is nowhere near as mature as
Hibernate, and indeed, we truly doubt it ever will. The purpose of the SPOD system is not to provide
true object to relational mapping, but to provide a platform that you can extend in your own applications
where necessary.
Chapter 11: Performing Server-Side Trickery

**Getting Started**

As always, the best way to learn how to use a tool is to jump straight in and use it. With each use of the SPOD, you will always require at least one class that extends the `neko.db.Object` class and one that either is or extends the `neko.db.Manager` class. The following example shows how the `neko.db.Object` class might be extended.

**Simple Use of the SPOD System**

This is the class that maps to the Author database table:

```pascal
class Author extends neko.db.Object
{
    // field reference variables
    public var int_id : Int;
    public var var_username : String;
    public var var_password : String;
    public var var_email : String;
    public var dte_added : Date;
    // associated table name
    static var TABLE_NAME = "Author";
    // associated table id field
    static var TABLE_IDS = ["int_id"]; // manager for this class
    public static var manager = new neko.db.Manager<Author>(Author);
}
```

And the application class that consumes it:

```pascal
import Author;
class AuthorApp
{
    public static function main()
    {
        var dbloc = "Wrox.db3";
        getAuthor( dbloc );
    }
    public static function getAuthor( loc : String )
    {
        var conn = neko.db.Sqlite.open( loc );
        // initialize the neko.db.Manager class
        neko.db.Manager.cnx = conn;
        neko.db.Manager.initialize();
        // get the author with the id value 1
        var u = Author.manager.get(1);
        if( u == null ) {
            u = new Author();
            // set the id to 1
            u.int_id = 1;
            // populate remaining items
            u.var_username = "Neko";
            u.var_password = "beeswax";
```
Part II: Server Side, JavaScript, and Flash: Oh My!

(continued)

```javascript
u.var_email = "Neko@TheCat.co.jp";
u.dte_added = Date.now();
u.insert();
} else {
  // update
  u.dte_added = Date.now();
u.update();
}
neko.Lib.print( "name: " + u.var_username + ", pass: " +
  u.var_password + "\n" );
// close the connection
neko.db.Manager.cleanup();
conn.close();
```

The `Author` class is the class that defines the data contained in the Author table of the example SQLite database. The five field reference variables represent each of the columns in the table using the same name. This is important for the `Manager` class to be able to map the columns successfully. If you wish to provide alternative names for these variables for access outside of the class, then you could always use properties. The next two variables represent the table name and id fields respectively. The table id field exists in an array in case you want to specify more than one field as the id for that table. You must provide both of these variables before specifying the `Manager` class variable. Failure to provide them defaults the table name to the name of the class and the id field to the value `id`.

In the `AuthorApp` class, the `Manager` class is first passed an open connection object. It matters little which database this is for, as long as the database wrapper is SPOD ready. The actual processes performed on the database are relative to most, so no checking is required. From there, the `Manager` is initialized. This prepares various object containers ready for population from the database. You won’t access these containers directly, but they are still required by the `Manager` class in order to, well, manage.

From here, the `Manager` class is ready to accept requests. In this particular instance, you have used the `neko.db.Manager.get` method, which retrieves a record based on its id and returns it in its mapped object type, provided the id field is of a type that can map freely to the haXe integer type. If the type of the id field is not mappable to an integer or multiple fields are required for the id, then the `neko.db.Manager.getWithKeys` method may be used instead. Both of these methods will be discussed in a little more detail, later.

From this point on, you are free to modify and use the object as you would any other object, with the exception that you are provided with several methods that provide a direct link to the data contained in the object with the data contained in the database. As you can see in the example, the `insert` and `update` methods may be used depending on the availability of a record with the given id.

Finally, when all your work has completed, the `Manager` class’s `cleanup` method is called and the database connection closed. `Manager.cleanup` empties several containers of data in order to free up memory and other resources.
Chapter 11: Performing Server-Side Trickery

**Table Mapping with neko.db.Object**

The `neko.db.Object` class is merely a container providing specific static variables and methods that call associated versions of themselves in the `neko.db.Manager` class. The point of `neko.db.Object` is to help in providing a base class for your objects that are compatible with the `neko.db.Manager` functionality and to help in its effort to correctly map your object to its relative table.

Table 11-4 details the methods of the `neko.db.Object` class, while Table 11-5 details its static variables. All of the methods listed in Table 11-4 do not accept any parameters.

### Table 11-4

<table>
<thead>
<tr>
<th>Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new()</code> : Void</td>
<td>Class constructor.</td>
</tr>
<tr>
<td><code>delete()</code> : Void</td>
<td>Deletes the record associated with the current object from the database.</td>
</tr>
<tr>
<td><code>insert()</code> : Void</td>
<td>Inserts the data associated with the current object into a new record in the database.</td>
</tr>
<tr>
<td><code>sync()</code> : Void</td>
<td>Re-populates the current object from the database.</td>
</tr>
<tr>
<td><code>ToString()</code> : String</td>
<td>Displays the object and table information as a string.</td>
</tr>
<tr>
<td><code>update()</code> : Void</td>
<td>Updates the database with the data stored in the current object.</td>
</tr>
<tr>
<td><code>RELATIONS()</code> : [{key : String, prop : String, manager : Manager}]</td>
<td>An optional static function returning a list of objects depicting the relationship between the current object and other objects. The <code>RELATIONS</code> function requires that the class object also contains a variable store with property accessor and Manager instance.</td>
</tr>
</tbody>
</table>

### Table 11-5

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>local_manager : neko.db.Manager&lt;neko.db.Object&gt;</code></td>
<td>Provides an instance of the <code>neko.db.Manager</code> class so as to provide method access from each of the <code>neko.db.Object</code> methods.</td>
</tr>
<tr>
<td><code>TABLE_NAME : String</code></td>
<td>Provides the name of the table associated with the object. Default is the class name.</td>
</tr>
<tr>
<td><code>TABLE_IDS : Array&lt;String&gt;</code></td>
<td>Provides the id field or fields. Default is id.</td>
</tr>
<tr>
<td><code>PRIVATE_FIELDS : Array&lt;String&gt;</code></td>
<td>Specifies the fields in the object that do not map to the relative database table.</td>
</tr>
<tr>
<td><code>manager : Neko.db.Manager</code></td>
<td>Specifies the class extending <code>neko.db.Manager</code> for your object.</td>
</tr>
</tbody>
</table>
Part II: Server Side, JavaScript, and Flash: Oh My!

**The TABLE_NAME and TABLE_IDS Variables**

The `TABLE_NAME` and `TABLE_IDS` are static variables that provide the SPOD framework with the means to map the data contained in your custom objects with data in the database. The `TABLE_NAME` variable provides a string to the `Manager` object that enables it to call the correct table when mapping data. If you fail to specify the `TABLE_NAME` variable, then the class name of your custom object is used. The `TABLE_IDS` variable, however, expects an array of strings listing the fields for that table that make up its unique id. When dealing with a SQLite database, you could quite easily pass the value `"OID"`, which would tell the SPOD to use the bespoke SQLite row id for that table.

**The PRIVATE_FIELDS Variable**

Each of the objects you create will provide variables with the same names as the fields in the database tables they map. When updating or inserting values into a database table using one of these objects, the `Manager` class loops through these variables and attempts to insert the contained data into fields of the same name. Now, chances are, you’re going to want to add other variables to your objects that do not map to database fields. If so, then you need to list those variable names in a static string array called `PRIVATE_FIELDS`, so that the `Manager` object knows not to include these values. For instance, given your current Author table, you may have an object used for mapping that also provides a couple of arrays used for other purposes. In this case, you need to make sure they are listed in a static `PRIVATE_FIELDS` variable:

```javascript
class Author extends neko.db.Object {
    // field reference variables
    public var int_id : Int;
    public var var_username : String;
    public var var_password : String;
    public var var_email : String;
    public var dte_added : Date;

    // custom variables
    public var chaptersWritten : Array<String>;
    private var copiesSold : Int;

    // associated table name
    static var TABLE_NAME = "Author";
    // associated table id field
    static var TABLE_IDS = ["int_id"];

    // list our non-field variables
    static var PRIVATE_FIELDS = ["chaptersWritten", "copiesSold"];

    // manager for this class
    public static var manager = new neko.db.Manager<Author>(Author);
}
```

As you can see, it doesn’t matter whether the variables you specify are public or private. Each and every variable that doesn’t map to a database field should be listed here. The one exception to this rule is for variables that act as keys for tables linked using the `RELATIONS` method.
Chapter 11: Performing Server-Side Trickery

**delete(), insert(), and update()**

Any object extending `neko.db.Object` is a container for a single record from the specifically mapped table. This provides a unique opening for being able to work with individual records at a time and then save that information within the database when required.

When working with an instance of the object already containing data from the database, you can choose to delete it from the database in its entirety by calling the object's `delete` method. Once performed, the object will still contain the original data from that record, including the unique identifier. Therefore, it is probably wise to delete the object, also, once this method has successfully returned.

Upon modifying data contained in the object, you can update the database to match the content by calling the `update` method.

Both the `update` and `delete` methods require that the object in question has already received data from the database. If not, then a missing key exception is thrown, as, even if you have prepopulated the object yourself, the unique id for that object would not be available. Also, any call to `update` requires that the original object be already in the `Manager` data cache, so it can iterate through all available fields and update only the data that has been modified since it was last synchronized with the database.

The `insert` method will work regardless of whether the object contains data from the database, or data you have manually entered. If you do not manually enter a value for the unique key, then it will be automatically populated after the `insert` using the `lastInsertedId` method for that table. At the same time, if you are inserting from an object that already contains a valid identifier from an existing record, you should set the value to `null` before calling the `insert` method, so that the value used is that of an automatic increment of the largest valid identifier and ensuring the insert is more likely to succeed.

**The sync() Method**

While the `update` method provides a means to synchronize data by submitting changes from the object to the database, the `sync` method provides the same facility, only in reverse. When called, the object will be repopulated with data from its associated record, overwriting any changes made since its last synchronization. This is very handy should your object contain dirty data (data not yet committed to the database) that you do not want persisted.

Another use for this method is when inserting data into a table that provides default values. If, when manually populating your object, you choose to leave some data population to the database, any call to the `insert` method will still mean that those unmodified fields contain a `null` value, even if your newly inserted record does not. Calling the `sync` method at such times will populate in the reverse direction, ensuring that object and record contain complete and identical values.

**The neko.db.Manager Class**

The `neko.db.Manager` class provides an extensive array of tools for dealing with object mapped database tables beyond what you've already seen. To begin, you are not restricted to dealing with objects that can only call a single record from a database, but can deal with many by using the `Manager` class's various array of tools. Table 11-6 details the methods of the `neko.db.Manager` class.
**Table 11-6**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new( obj : Dynamic ) : Void</td>
<td>Accepts the class of the parent <code>neko.db.Object</code> object as the parameter and the type of the parent <code>neko.db.Object</code> as the type. See below.</td>
</tr>
<tr>
<td>all( val : Bool ) : List&lt;T&gt;</td>
<td>Returns all the records of the associated table within a list of the type passed in the <code>Manager</code> class declaration.</td>
</tr>
<tr>
<td>count() : Int</td>
<td>Returns the number of records in the table.</td>
</tr>
<tr>
<td>doDelete( val : T ) : Void</td>
<td>Deletes the record contained in the passed SPOD object.</td>
</tr>
<tr>
<td>doInsert( val : T ) : Void</td>
<td>Inserts data from the data contained in the passed SPOD object.</td>
</tr>
<tr>
<td>doSync( val : T ) : Void</td>
<td>Synchronizes the data in the passed object from its associated record.</td>
</tr>
<tr>
<td>doUpdate( val : T ) : Void</td>
<td>Updates data in the database from the data contained in the passed SPOD object.</td>
</tr>
<tr>
<td>get( id : Int, lock : Bool ) : T</td>
<td>Gets a record specified by its unique integer identifier and returns a SPOD object.</td>
</tr>
<tr>
<td>GetWithKeys( keys : Dynamic, lock : Bool ) : T</td>
<td>Gets a record specified by one or more fields representing the record identifier and returns a SPOD object.</td>
</tr>
<tr>
<td>object( sql : String, lock : Bool ) : T</td>
<td>Executes a given SQL string and returns a SPOD object.</td>
</tr>
<tr>
<td>objectToString( spod : T ) : String</td>
<td>Converts a given SPOD object to a string.</td>
</tr>
<tr>
<td>objects( sql : String, lock : Bool ) : List&lt;T&gt;</td>
<td>Executes a given SQL string and returns a list of SPOD objects.</td>
</tr>
<tr>
<td>quote( val : String ) : String</td>
<td>Escapes quotes in a string for use in a SQL string.</td>
</tr>
<tr>
<td>result( sql : String ) : Dynamic</td>
<td>Returns a non-SPOD object from a given SQL string. Works in the same way as using the traditional ResultSet class.</td>
</tr>
<tr>
<td>results&lt;T&gt;( sql : String ) : List&lt;T&gt;</td>
<td>Returns a list of non-SPOD objects from a given SQL string. Works in the same way as using the traditional ResultSet class.</td>
</tr>
<tr>
<td>search( keys : Dynamic, lock : Bool ) : List&lt;T&gt;</td>
<td>Returns a list of SPOD objects from a series of conditions held as key/value pairs.</td>
</tr>
</tbody>
</table>
# Chapter 11: Performing Server-Side Trickery

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cleanup() : Void</td>
<td>Releases all data held in the <code>neko.db.Manager</code> class’s global cache.</td>
</tr>
<tr>
<td>initialize() : Void</td>
<td>Prepares the <code>neko.db.Manager</code> class for use.</td>
</tr>
<tr>
<td>make( val : T ) : Void</td>
<td>This method, which can be overridden, is called just after class initialization.</td>
</tr>
<tr>
<td>unmakemake( val : T ) : Void</td>
<td>This method, which can be overridden, is called just before a data update.</td>
</tr>
<tr>
<td>select( sql : String ) : String</td>
<td>Builds a SQL string with a passed condition string construct appended to the end and also a <code>FOR_UPDATE</code> command.</td>
</tr>
<tr>
<td>selectReadOnly( sql : String ) : String</td>
<td>Builds a SQL string with a passed condition string construct appended to the end.</td>
</tr>
</tbody>
</table>

As you can see, the functionality provided by the `neko.db.Manager` class is pretty extensive. All manner of methods exist for retrieving and modifying data, whether returned in a SPOD-compatible object or not. Given the previous example, you could quite easily rewrite the code to allow for multiple records to be returned by using the `all` method, and then iterating through the contents of the returned `List`:

```java
import Author;
class AuthorApp
{
  public static function main()
  {
    var dbloc = "Wrox.db3";
    getAuthor( dbloc );
  }
  public static function getAuthor( loc : String )
  {
    var conn = neko.db.Sqlite.open( loc );
    neko.db.Manager.cnx = conn;
    neko.db.Manager.initialize();
    // get the author with the id value 1
    var users = Author.manager.all( true );
    for ( u in users ) {
      if ( u == null ) {
        u = new Author();
        // set the id to 1
        u.int_id = 1;
        // populate remaining items
        u.var_username = "Neko";
        u.var_password = "beeswax";
        u.var_email = "Neko@TheCat.co.jp";
      }
    }
  }
}
```

(continued)
As you can see, little was required to facilitate this change, and yet suddenly, you have access to the entire contents of the table as a list of objects. With minimal effort, you could extract whatever data you saw fit while still adhering to an object mapping system. For example, you might want to extract only those authors who have the letter e in their name:

```javascript
Author.manager.objects( Author.manager.select("var_username LIKE '%e%'"), true );
```

**Extending the neko.db.Manager Class**

Extending the neko.db.Manager class for a specific table or group of tables opens up a whole new world of possibilities. For one, the Manager subclass could be filled with numerous custom record retrieval methods that you might need to use regularly. Also, the make and unmake methods called at key times within the Manager class code provide a great way to ensure data integrity. Such calls provide pseudo trigger functionality that you can use to handle data between trips to the database. You could even extend the existing functions within the Manager class and provide your own event methods.

When extending the neko.db.Manager class, you need to make sure you forward the relevant neko.db.Object subclass to the parent constructor. Once this is done, you can pretty much do what you like. For instance, here is an example that facilitates a method returning authors with a given e-mail address:

```javascript
class AuthorManager extends neko.db.Manager<Author> {
    public function new() {
        super( Author );
    }
    public function getByEmail( email : String ) {
        return objects(select("var_email = "+quote( email ) ), true );
    }
}
```

By supplying various methods to the extended Manager class, you are in affect creating application-based stored procedures. This is a very handy feature when using a database with limited capabilities, such as a SQLite database.
Chapter 11: Performing Server-Side Trickery

**make() and unmake()**

The `make` and `unmake` methods are override-able functions for use in extensions of the `neko.db.Manager` class. The `make` method is called when a returned SPOD object is created following data retrieval from the database, while the `unmake` method is called just before an update is made. Using both methods together, you can purposely perform calculations on data retrieved from the database so as to make the data more object friendly, while using the `unmake` method to reparse the data for saving back to the database:

```javascript
class AuthorManager extends neko.db.Manager<Author> {
    public function new() {
        super( Author );
    }
    private function make( u : Author ) {
        return Author.favoriteFoodsArray = Author.faveFoods.split(',');
    }
    private function unmake( u : Author ) {
        return Author.faveFoods = Author.favoriteFoodsArray.join(',',$);
    }
}
```

**Global Cache**

Every SPOD object returned from the database is added to a global cache stored in the associated `neko.db.Manager` object. The purpose of this cache is to facilitate data consistency, so that, should two SPOD objects both contain records from the same table with the same unique identifier, then modifying the data in one object will automatically modify the data of the second.

**Adding Table Relations**

As you’ve seen from the `neko.db.Object` class methods, the SPOD system supports assigning table relations. Now, this is pretty basic, but still useful. The idea is you can include a call to a single record in a table where the id of that record exists in the primary record being called. This can be perpetuated; so, for example, you can have one record link to another, which links to another, and so on.

Take a look at how this is done. First, you need to create a secondary table to link to, and alter the first table to include a field that will contain the second table’s unique identifier:

```javascript
class CreateTableChapters
{
    public static function main()
    {
        var dbLoc : String = "Wrox.db3";
        var dbFactory = create( dbLoc );
    }
    public static function create( loc : String )
    {
        var cnx = neko.db.Sqlite.open( loc );
        cnx.request( "CREATE TABLE Chapter ( c11.indd   295c11.indd   295 12/27/07   10:02:01 AM12/27/07   10:02:01 AM
```
Now that the tables are sorted, they need to be mapped. The class for the Author table already exists; it just needs to be modified to include the relationship data:

```javascript
import Chapter;
class Author extends neko.db.Object {
    public var int_id : Int;
    public var var_username : String;
    public var var_password : String;
    public var var_email : String;
    public var int_chapter_id : Int;
    public var dte_added : Date;
    static function RELATIONS() {
        return [{ prop : "chapter", key : "int_chapter_id", manager : untyped Chapter.manager }];
    }
    static var TABLE_NAME = "Author";
    static var TABLE_IDS = ["int_id"];  
    private var chapterId : Int;
    public var chapter(dynamic,dynamic) : Chapter;
    public static var manager = new neko.db.Manager<Author>(Author);
}
```

And the mapping for Chapter:

```javascript
class Chapter extends neko.db.Object {
    public var int_id : Int;
    public var int_chapter : Int;
    public var var_title : String;
    static var TABLE_NAME = "Chapter";
    static var TABLE_IDS = ["int_id"];  
    public static var manager = new neko.db.Manager<Chapter>(Chapter);
}
```

As you can see, the Chapter class isn’t all that different from the original Author class, in terms of features. The biggest difference shown in these classes are the lines that link the Author class to the Chapter class.
Chapter 11: Performing Server-Side Trickery

The following class creates a new record in the Chapter table for an author with the unique id of 1. If the record already exists, then the data for that record is displayed. The point to take note of is that the data is called through the instance of the Author class, not the Chapter class:

```java
import Author;

class AuthorChapterApp {
    public static function main()
    {
        var dbloc = "Wrox.db3";
        getAuthor( dbloc );
    }

global static function getAuthor( loc : String )
{
    var conn = neko.db.Sqlite.open( loc );
    neko.db.Manager.cnx = conn;
    neko.db.Manager.initialize();
    // get the author with the id value 1
    var u = Author.manager.get(1);
    if( u != null )
    {
        if( u.chapter == null )
        {
            var c = new Chapter();
            // populate new chapter
            c.int_chapter = 11;
            c.var_title = "Server Side Trickery";
            c.insert();
            u.chapter = c;
            u.update();
            neko.Lib.print( "chapter updated" );
        } else {
            neko.Lib.print( "chapter : " + u.chapter.int_chapter + ", " + u.chapter.var_title );
        }
    } else {
        // close the connection
        neko.db.Manager.cleanup();
        conn.close();
    }
}
}
```

**The RELATIONS() Method**

The static RELATIONS method provides one purpose; to return an array containing objects detailing the relationship between the table represented by the method’s parent class and other tables represented by other objects extending the neko.db.Object class. Each object must specify the fields as detailed in Table 11-7. Aside from the RELATIONS method, you also need to supply the getters and setters detailed in each object in the array returned from the RELATIONS method.
Table 11-7

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prop</td>
<td>The name of a dynamic getter and setter used to retrieve a reference to the object containing the data from the secondary table</td>
</tr>
<tr>
<td>key</td>
<td>The name of the field from the primary table containing the unique id of the record to retrieve from the secondary table</td>
</tr>
<tr>
<td>manager</td>
<td>The Manager object used to retrieve the secondary table data</td>
</tr>
</tbody>
</table>

When assigning objects through reference to an existing record, the object you assign must also represent an existing record. In the previous example, the newly created Chapter record was assigned to the Author object after it was inserted into the database. This is important, as the unique identifier of the Chapter object is required to successfully update the Author object’s associated record.

**Complex Table Relations**

Performing complex relations between tables, such as one-to-many and many-to-many relations, is possible, though it requires that you build some essential SQL statements yourself. The necessary functionality for performing such feats automatically does not yet exist within the haXe SPOD system, and possibly never will. However, with a little fiddling and much trial and error, some workarounds can be achieved.

In order to realize complex relations using the SPOD system, you need to look at using the `options` method of the `neko.db.Manager` class, called in the `make` method for that object. Essentially, creating a one-to-many relationship should be no different from a many-to-many relationship. The main obvious difference is that you would initially have to cater for a list of SPOD objects within the application class rather than a solitary SPOD object. The trick is to actually create a one-to-many query, but have each received record make a one-to-many query in the opposite direction, calling all the original classes, thus creating the many-to-many relationships. Now, normally you’d imagine this to cause some sort of infinite loop, with records everywhere calling each other. However, despite the number of references to a single record, providing each record possesses a unique id, the SPOD framework will only ever store one instance of each record in memory, thus providing the perfect relational object mapping.

As always, all of this will be better understood within an example. Assuming you want to create the many-to-many relationship, you first need to create a lookup table to link the Author and Chapter tables:

```neko
class CreateLookupTable
{
    public static function main()
    {
        var dbLoc : String = "Wrox.db3";
        var dbFactory = create( dbLoc );
    }

    public static function create( loc : String )
    {
        var cnx = neko.db.Sqlite.open( loc );
    }
}
```
Chapter 11: Performing Server-Side Trickery

```plaintext
cnx.request( "CREATE TABLE Lookup (  int_id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  int_author INTEGER NOT NULL,  int_chapter INTEGER NOT NULL )" );
cnx.request( "INSERT INTO Lookup (  int_author, int_chapter ) VALUES ( 1, 1 )" );
cnx.request( "INSERT INTO Lookup (  int_author, int_chapter ) VALUES ( 1, 2 )" );
cnx.close();

This new table stores references to records from both tables, mapping each author to one or more chapters and possibly each chapter to one or more authors. As you can see, once the table is created, it’s populated with a couple of records, so that you have something to test with.

Next, you need to rewrite the Author table to facilitate the storage of a list of Chapter objects:

```plaintext
class Author extends neko.db.Object
{
    public var int_id : Int;
    public var var_username : String;
    public var var_password : String;
    public var var_email : String;
    public var dte_added : Date;
    public var int_chapter_id : Int; // Now deprecated
    public var chapters : List<Chapter>;
    static var PRIVATE_FIELDS = ["chapters"];
    static var TABLE_NAME = "Author";
    static var TABLE_IDS = ["int_id"];
    public static var manager = new AuthorManager();
}
```

As you can see, the int_chapter_id will now be deprecated. This is because the Lookup table will now facilitate all references to an author’s chapter records. Another minor change is the inclusion of the chapters field, which holds a reference to all the associated chapter objects, while a fairly major change is the disappearance of the relations table. Because you no longer directly relate the tables, but instead provide the functionality in a callback method, the standard SPOD relationship functionality is now unusable.

The Chapter object should now look very similar, as it will need to be able to perform the reverse queries:

```plaintext
class Chapter extends neko.db.Object
{
    public var int_id : Int;
    public var int_chapter : Int;
    public var var_title : String;
    public var authors : List<Author>;
    static var PRIVATE_FIELDS = ["authors"];
}
```

(continued)
Part II: Server Side, JavaScript, and Flash: Oh My!

(continued)

```java
static var TABLE_NAME = "Chapters";
static var TABLE_IDS = ["int_id"];
public static var manager = new ChapterManager();
```

The callback methods needed to perform the calls to the associated record objects need to be performed in an object that extends the `neko.db.Manager` class. You need to provide two of these: one for the `Chapter` class and one for the `Author` class. Again, because of the reverse data queries, these two classes should look very similar. Here is the `AuthorManager` class:

```java
class AuthorManager extends neko.db.Manager<Author>
{
    public static var cmgr : ChapterManager;
    public function new()
    {
        super(Author);
        ChapterManager.amgr = this;
    }
    function make( a : Author ) : Void
    {
        a.chapters = cmgr.getAuthorChapters( a.int_id );
    }
    public function getChapterAuthors( c : Int ) : List<Author>
    {
        return objects( "select distinct u.* from Lookup l inner join
                        Author u on l.int_author = u.int_id where l.int_chapter = "
                        + c, true );
    }
}
```

And the `ChapterManager` class:

```java
class ChapterManager extends neko.db.Manager<Chapter>
{
    public static var amgr : AuthorManager;
    public function new()
    {
        super(Chapter);
        AuthorManager.cmgr = this;
    }
    function make( c : Chapter ) : Void
    {
        c.authors = amgr.getChapterAuthors( c.int_id );
    }
    public function getAuthorChapters( a : Int ) : List<Chapter>
    {
        return objects( "select distinct c.* from Lookup l inner join
                        Chapters c on l.int_chapter = c.int_id where l.int_author = "
                        + a, true );
    }
}
```
Chapter 11: Performing Server-Side Trickery

So, what do these two classes do? Well, first they map the specific record object type and class to the underlying `Manager` class functionality. One could supply a T value, which represents a template, meaning that you could supply the values for the type and class when instantiating the class. However, seeing as the data queries are hard coded, the associated data type will be unlikely to change.

Once the classes are instantiated and the underlying functionality is initialized, each class specifies a reference to the other class in a static variable. Now, this took some figuring out when we first performed a many-to-many query using the SPOD system, though it now seems to work very well. It appears that, when the `make` method is called following a query for the type associated with the `Manager` object, the aforementioned type is then bound to any queries made using the `Manager` methods. This then means that, when you want to make a joined query call to the records from a foreign table using, say, the `objects` method, the SPOD system will still try to map the returned data to the initial type. So, by providing a reference to the other class, you can then make method calls to that class and return the data in that class’s associated type; a sort of you scratch my back and I’ll scratch yours scenario.

One would then normally expect this to go on for some time in some form of recursive loop. As an author record is pulled, it fires several chapter queries, which in turn fires more author queries and so on. However, as mentioned previously, this will not happen for any records already existing in the SPOD cache. Should a record with a previously mapped id be returned from a query, its data will be discarded in favor of the stored data.

Working with Files

As with its database support, the haXe language also provides a decent array of tools for accessing and manipulating the filesystem. Utilizing streams, haXe makes it all too easy to direct file data wherever you want, or direct almost any external data into a file. For instance, utilizing the haXe FTP functionality from the Motion-Twin libraries, you could quite easily bind the handle from a `File` object and redirect its data directly into a stream attained from a remote file, thus facilitating a transfer of file content, rather than the file itself.

Traversing Files and Directories

File and directory traversal functionality in Neko can be accessed from the `neko.FileSystem` class. Here, you will find common functionality used in locating and describing files as well as complete directory management methods. Table 11-8 details the methods of the `neko.FileSystem` class:

<table>
<thead>
<tr>
<th>Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createDirectory(path : String) : Void</code></td>
<td>Creates a directory from the given string</td>
</tr>
<tr>
<td><code>deleteDirectory(path : String) : Void</code></td>
<td>Deletes a specified directory</td>
</tr>
<tr>
<td><code>deleteFile(path : String) : Void</code></td>
<td>Deletes a specified file</td>
</tr>
</tbody>
</table>

Table continued on following page
Part II: Server Side, JavaScript, and Flash: Oh My!

Paths

One of the key features when dealing with files in Neko is its level of forgiveness with regard to path strings. Neko is one of the few languages that does not have a nervous breakdown because you chose to include backslashes and forward slashes, as well as a couple of parent directory signifiers, throughout a path string. Certainly, something we very much welcome. For example, on our hard drive, we could reference the *FileSystem* class file like this:

```javascript
var path = "C:\haxe\std\../../haxe\std\neko/FileSystem.hx";
```

As you can see, it’s pretty messy, but sometimes you can really benefit from this level of leniency. Especially when dealing with paths constructed with functions.

Another great feature is being able to quickly change between relative and absolute paths with ease. As you can probably see, the preceding example is kind of both absolute and relative, which can be lifesaving when building distributed applications, where your level of interaction with the end user machine may be somewhat limited.

**fullPath()**

The *fullPath* method of the *neko.FileSystem* class provides the means to directly provide an absolute path representation of a given relative path. This is not normally necessary when used in combination with the majority of Neko’s libraries, but may prove useful for data logging and the like. To use it, you simply pass it a string depicting a relative path, and it will return the full path equivalent if possible:

```javascript
var abs : String = neko.FileSystem.fullPath( "../data" );
```

---

### Methods Description

<table>
<thead>
<tr>
<th>Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exists( path : String ) : Bool</code></td>
<td>Checks for the existence of a specified file or directory</td>
</tr>
<tr>
<td><code>fullPath( rel : String ) : String</code></td>
<td>Provides an absolute path from a path relative to the application</td>
</tr>
<tr>
<td><code>isDirectory( path : String ) : Bool</code></td>
<td>Returns <code>true</code> if the specified path is a directory; otherwise false is returned</td>
</tr>
<tr>
<td><code>kind( path : String ) : neko.FileKind</code></td>
<td>Returns an <em>enum</em> value based on whether the passed path is a file or directory</td>
</tr>
<tr>
<td><code>readDirectory( path : String ) : Array&lt;String&gt;</code></td>
<td>Returns the items found in a directory</td>
</tr>
<tr>
<td><code>rename( path : String, new : String )</code></td>
<td>Renames a file or directory to the given new name</td>
</tr>
<tr>
<td><code>stat( path : String ) : neko.FileStat</code></td>
<td>Returns various data for a given path</td>
</tr>
</tbody>
</table>
exists()

The `exists()` method is probably the method we personally use most among the filesystem arsenal. With it, you can determine if a given path, whether absolute or relative, actually exists. This is necessary if you want to directly access a file or directory, without causing an exception if it isn’t where you expected:

```java
if (!neko.FileSystem.exists( "../images" ))
    neko.FileSystem.createDirectory( "../images" );
```

Directory and File Management

Directory management in Neko is not exactly overflowing with functionality, but it does provide the basic necessary methods for your general requirements, while file management covers several classes as you’ll see a little later. At present, there aren’t any methods provided for `chmod` functionality, without resorting to accessing system commands outside of the Neko framework. Also, a method for moving files and directories is also missing. With luck, though, these methods will be added in a later release of the haXe framework.

createDirectory(), deleteDirectory(), and deleteFile()

Creating a directory is easy, because a directory either is or it isn’t. There are no types of directory, unless you start delving into icons and whatnot, which you’re not going to do. To create a directory, you use the `createDirectory` method. Simple, huh? `createDirectory` only accepts one parameter; the path of the directory you want to create. The only restrictions to this method are that you have the relevant permissions to create the directory in the specified location, and that the directory you want to create is in an existing directory. `createDirectory` will not create more than one directory at a time:

```java
neko.FileSystem.createDirectory( "c:\projects" );
```

Deleting a directory follows the same structure. Again, the specified path must provide a valid directory with the necessary permissions:

```java
neko.FileSystem.deleteDirectory( "c:\projects" );
```

And deleting a file:

```java
neko.FileSystem.deleteFile( "c:\projects\archive.zip" );
```

Reading from a Directory

Okay, so you can create directories and delete them, but you might find it hard to manage directories if you don’t know what they contain. Thankfully, Neko provides a handy `readDirectory` function for listing the contents of a directory as an array of strings:

```java
var contents : Array<String> = neko.FileSystem.readDirectory( "../images" );
```

Once you have your string array, you need to work out whether each item is a directory or a file. This can be done in one of two ways. First, you can use the `isDirectory` method. This method expects a string representing the path to the file or directory you want to query and will return a Boolean value; `true` if the path is a directory and `false` if it isn’t. If the path leads to a file or directory that does not exist, then an exception will be thrown.
Part II: Server Side, JavaScript, and Flash: Oh My!

The second option is to use the kind method. This method will return kdir of the neko.FileKind enumerator if the path is a directory, kfile if it's a file, or will throw an exception if the path does not exist. A third enumerator, kother, may also be provided should the specified path be neither a file nor a directory.

Working with Files and Streams

Working with file creation, along with writing data to and from files, requires the use of the neko.io package. There are a good number of classes here, including several base classes you can extend to provide your own stream handling functionality. All data written to files comes from the base neko.io.FileOutput class, while reading from a file may or may not require the use of the neko.io.FileInput class. Table 11-9 lists the methods provided by the neko.io.File class.

**Table 11-9**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append( path : String,</td>
<td>Returns an output stream object for appending data to the end of a file.</td>
</tr>
<tr>
<td>binary : Bool ) : FileOutput</td>
<td></td>
</tr>
<tr>
<td>copy( src : String, dst :</td>
<td>Copies a file from one location to another. Copying a directory throws an exception.</td>
</tr>
<tr>
<td>String ) : Void</td>
<td></td>
</tr>
<tr>
<td>getChar( echo : Bool ) : Int</td>
<td>Reads a char value from the associated file.</td>
</tr>
<tr>
<td>getContent( path : String ) : String</td>
<td>Returns the string content from a given path.</td>
</tr>
<tr>
<td>read( path : String, binary : Bool ) : FileInput</td>
<td>Returns an input stream object for reading data from a file.</td>
</tr>
<tr>
<td>stderr() : FileOutput</td>
<td>Returns an output stream object for appending data to the stderr object.</td>
</tr>
<tr>
<td>stdin() : FileInput</td>
<td>Returns an input stream object for reading data from the stdin object.</td>
</tr>
<tr>
<td>stdout() : FileOutput</td>
<td>Returns an output stream object for appending data to the stdout object.</td>
</tr>
<tr>
<td>write( path : String, binary : Bool ) : FileOutput</td>
<td>Returns an output stream object for writing data to a file.</td>
</tr>
</tbody>
</table>

Reading and Writing Data to a File

Thanks to the stream objects provided by the neko.io package, it's possible to access the content of a file no matter where that file is located. All that is required is a FileInput or FileOutput handle on the file in question. Once a file stream handle is assigned to a file object, you can theoretically output the data.
from that file to pretty much anywhere, including a remote server, if you so want. Likewise, any data you can push into a stream object can be directed into the content of a file.

To attain a stream handle on a file, you first have to decide which direction your data will be traveling. For example, if you wish to stream data out of the file, then you need to create a `FileInput` stream object, while the `FileOutput` stream will write to the file. Personally, we often get confused over which class is which, as to me, `FileInput` would assume inputting data into the file, though you could look at this either way.

In terms of syntax, it matters little whether you are creating a stream for writing or for reading. Both methods require a similarly structured method call and can be used in the same fashion. This makes sense, as one file’s output could quite possibly become another file’s input.

**Reading Data from a Stream**

To create a `FileInput` stream from a file, you use `neko.io.File.read`. This method accepts the path to the file as a string and a Boolean depicting whether or not the file contains binary data:

```haxe
var stream = neko.io.File.read( "pathToMyFile.zip", true );
```

**readChar**

Once you have your returned `FileInput` stream, you then have full control over where you direct its throughput. For instance, to return each character from the file one by one, you would use `readChar`. This method returns the characters as an integer value, which you can convert to a string character using `Std.chr`. In this example, the returned character values are simply appended to the end of a haXe string:

```haxe
class ReadChar
{
    public static function main()
    {
        var stream = neko.io.File.read( 'example.txt', false );
        var str = "";
        while ( !stream.eof() )
        {
            try
            {
                str += Std.chr( stream.readChar() );
            }
            catch ( err : Dynamic)
            {
                //neko.Lib.print( err );
            }
            neko.Lib.print( str );
            stream.close();
        }
    }
}
```

As you can see, each `readChar` call exists within a `try…catch` block. This is because of the fact that when the stream reaches the end of file — or `eof` — it throws an exception. The problem is you don’t
Part II: Server Side, JavaScript, and Flash: Oh My!

know it's going to reach the end of file until it arrives, so you can only deal with it by trapping the error. The test for the end of file in the while clause prevents an infinite loop caused by the try...catch block not releasing the exception.

**readBytes**

The second way to read characters from a stream is by extracting them using the `readBytes` method. This method accepts three parameters; a buffer, the starting location in the buffer to write data, and the length of bytes to extract:

```javascript
var charactersRead = stream.readBytes( strBuffer, intStart, intLength );
```

The buffer is a string of a preset length, which is passed to the `readBytes` method and used as a container for the bytes you want to extract. You can use an existing string for this, or simply use the `makeString` method in the `neko.Lib` class. It is passed by reference, so once you have made a call to the `readBytes` method, you will then likely use the string elsewhere in the application, or store it in another medium.

When specifying the start location and length of bytes within the buffer, it is important to make sure that the sum of both values is not greater than the length of the buffer. Failure to enforce this will cause the `readBytes` method to fail. If you perform this method call within a while loop and such an event occurs, the stream will never reach an end-of-file state; thus the loop will perpetuate until broken by other means:

```javascript
class ReadBytes
{
    public static function main()
    {
        var stream = neko.io.File.read( "example.txt", false );
        var str = neko.Lib.makeString( 20 ); // 20 is the number of chars
        while ( !stream.eof() )
        {
            try
            {
                stream.readBytes(str, 0, 20);
            }
            catch ( err : Dynamic )
            {
                neko.Lib.println( err );
            }
            neko.Lib.println( str );
        }
        stream.close();
    }
}
```

Normally, when extracting bytes from a stream, you will need to check how many bytes were extracted using the return value of `readBytes`. This way, you can make use of the number of bytes returned, while avoiding the gobbledygook that preexisted in the string buffer. When finished with the stream, always remember to close it, so as to release the file handle and free valuable resources.

**readAll**

You can actually go one better than this, now, using the `readAll` method. `readAll` works similarly to the `getContent` method of the `neko.io.File` class, at least in terms of its general effect. Under the
hood, and in terms of its usefulness, the `readAll` method is completely different. The `getContent` method is explained next.

As you can probably guess, the `readAll` method allows the application to read all bytes from a stream. If necessary, you can supply an optional parameter specifying the maximum size of the buffer you are streaming to, but under most circumstances, this shouldn’t be necessary.

```java
class ReadAll
{
    public static function main()
    {
        var stream = neko.io.File.read( 'example.txt', false );
        var str : String = stream.readAll();
        stream.close();
        neko.Lib.println( str );
    }
}
```

### Directly Reading Data

Sometimes, you might consider reading data through the use of a stream object as a bit overkill. For instance, if you just want to read the contents of a text file for display in a web page or console. At such times, you are better off using the `getContent` method of the `File` class.

`getContent` is very simple. It accepts the path to the file you wish to query and returns a string containing the contents of the file:

```java
class GetContent
{
    public static function main()
    {
        var str = neko.io.File.getContent( "example.txt" );
        neko.Lib.print( str );
    }
}
```

As you can see, this certainly makes for lighter work, and is perfect for reading configuration files and the like.

### Writing Data to a Stream

There are two ways to attain a stream for writing, because there are two ways to write data. One way involves creating a new file using a given filename and writing content to it, while the other involves opening an existing file and appending content to the end of its existing data. Both are simple, and in many ways, not very different to reading from a stream. As there is so little difference to their usages, this chapter only details examples using one of the methods, though you can choose to use whichever method suits the task at hand.

To create a new file for writing, you would use:

```java
var stream = neko.io.File.write( "pathToFile.txt", boolIsBinary );
```
Part II: Server Side, JavaScript, and Flash: Oh My!

While appending to an existing file requires:

```javascript
var stream = neko.io.File.append( "pathToFile.txt", boolIsBinary );
```

The Boolean value in the second parameter should be set to false if the file will contain ASCII data, or true if it will contain binary.

**writeChar**

Once you have your `FileOutput` object, you can now choose to write bytes or characters to your stream. The `writeChar` method adds characters to the end of the stream data. It accepts a single parameter; the integer of the character to add:

```javascript
stream.writeChar( intChar );
```

To convert a character from a string to an integer, use the `Std.ord` method:

```javascript
class WriteChar
{
    public static function main()
    {
        var stream = neko.io.File.write( "example.txt", false );

        var str = "abcdefghijklmnopqrstuvwxyz";
        var pos = 0;
        while ( pos <= str.length )
        {
            try
            {
                stream.writeChar( Std.ord( str.charAt( pos ) ) );
            }
            catch ( err : Dynamic)
            {
                //neko.Lib.print( err );
            }
            pos++;
        }
        stream.close();
    }
}
```

Again, it is important that you close the stream after use, so as to free up valuable resources.

**writeBytes**

In this circumstance, the same feat can be accomplished more easily using the `writeBytes` method. If you know the length of the string you want to output through a stream, `writeBytes` enables you to dump the entire string content in one go, without having to perform a loop.

*When using `writeBytes`, it is possible for the buffer containing the data to not be fully output to the receiving device. In such circumstances, it is advisable to call the flush method, thereby forcing the buffer to clear and the data to be written to the device.*
class WriteBytes
{
    public static function main()
    {
        var stream = neko.io.File.write( "example.txt", false );
        var str = "abcdefghijklmnopqrstuvwxyz";
        try
        {
            stream.writeBytes( str, 0, str.length );
            stream.flush();
        }
        catch ( err : Dynamic)
        {
            //neko.Lib.print( err );
        }
        stream.close();
    }
}

write

As with reading from a stream, writing data to a stream also has a simple write-all approach, using the write method. write simply outputs an entire string to the given stream.

class WriteAll
{
    public static function main()
    {
        var stream = neko.io.File.write( "example.txt", false );
        var str = "abcdefghijklmnopqrstuvwxyz";
        try
        {
            stream.write( str );
            stream.flush();
        }
        catch ( err : Dynamic)
        {
            //neko.Lib.print( err );
        }
        stream.close();
    }
}

Traversing a Stream

When you are dealing with bytes in a stream, it is possible to move the current play head, as it were, to a different location for random byte access. haXe provides two methods to aid in this: tell and seek.

tell accepts no parameters and returns the current location of the marker in the stream as an integer:

    var position = stream.tell();

Seek, on the other hand, allows you to reposition the marker based on a given enumerator flag:

    stream.seek( intMove, seekCur );
Part II: Server Side, JavaScript, and Flash: Oh My!

The first parameter is the number of spaces you want to move the marker. You can use a positive number if you want the marker to move forward, or negative to move backward. The second parameter is a `neko.io.File.FileSeek` enumerator, which contains three possible values:

- SeekCur
- SeekBegin
- SeekEnd

`SeekCur` maps the bytes the marker should move from its current location in the stream. `SeekBegin` maps from the beginning of the stream, while `SeekEnd` maps from the end. For example, the following class writes a string containing the first ten letters in the alphabet:

```javascript
import neko.io.File;
class Seek
{
    public static function main()
    {
        var stream = neko.io.File.write( "example.txt", false );
        var str = "abcdefghijklmnopqrstuvwxyz";
        try
        {
            stream.writeBytes( str, 0, 10 );
            stream.seek( 3, SeekEnd );
            stream.writeBytes( str, 0, 10 );
            stream.seek( 3, SeekBegin );
            stream.writeBytes( str, 0, 10 );
            stream.seek( -5, SeekCur );
            stream.writeBytes( str, 0, 3 );
        }
        catch ( err : Dynamic)
        {
            //neko.Lib.print( err );
            stream.close();
        }
    }
}
```

If you run the code, you should receive an output `example.txt` file containing the string:

```
abcabcdeabcijabcdefgij
```

What’s happened is, the string `abcdefgij` was written to the stream. From there, the head was moved forward three places from the end of the stream, and the string was then rewritten, making the stream twenty-three characters in length. The head was then moved to the third character in the stream and the content was written again, leaving the marker at position thirteen. Finally, the marker was moved backward five places, and the string was once again written to the stream.
Chapter 11: Performing Server-Side Trickery

**Accessing a Remote File**

Earlier in this book, you looked at how to send data to a remote location using the `haxe.Http` class. In Neko, and indeed JavaScript, you can also use this class to request data. This is done via the `request` method, and like the `neko.io.File.getContent` method, it doesn't require streams:

```haxe
var string = haxe.Http.request( strURLofPageToRequest );
```

The `haxe.Http` method performs a blocking IO, so all code performed after the request will not occur until the request has completed.

**Creating a Logger Class**

Working with files in Neko proves immensely useful when combined with a logging facility. In this section, you'll now put your knowledge to work to extend the `Exception` class you built in Chapter 7 for use later in the book.

Let's examine the requirements for this:

- The `Logger` class should be accessible as a separate entity from the `Exception` class.
- The `Exception` class should have the full capability to write its own messages to the logger for output.
- The logger should be able to write to a file with a given name, or a file named with the current date.
- The logger should be extendable so that data can be written to alternate streams of your choosing.
- All data written to the logger should be time stamped.

Most applications have, at most, only one logger. There may be occasions when this is not so, but for the most part you could consider this as an important design point. Therefore, when designing the logger class, it should be necessary for much of the functionality to remain static so that all application classes may make use of it.

As any data provided by the `Exception` class will likely require more focus, functionality should exist to highlight log data of interest. One way this can be achieved would be to append customizable header and footer values to such content supplying a default string value. At this current stage, it is assumed that all data output from the logger will be stored in a sequential stream or file of some sort, but that might not always be the case, so alternative functionality should be easily appended to any output.

Begin with the data handling for the `Logger` class. To start things off, you're going to require some basic values:

```haxe
class Logger
{
    public static var useSingleLogFile : Bool = false;
    public static var logFileName : String = "Log";
    public static var location : String = "logs/";
    private static function getLogFileName()
%
```
Part II: Server Side, JavaScript, and Flash: Oh My!

(continued)

```javascript
var filename = logFileName;
if ( useSingleLogFile == false )
{
    var day = Date.now().getDate();
    var month = Date.now().getMonth();
    var year = Date.now().getFullYear();
    filename += "_" + day + "_" + month + "_" + year;
}
return filename + ".txt";
}
public static function header()
{
    return "\n\n********************";
}
public static function footer()
{
    return "********************\n";
}
public static function log( val : String, ?bold : Bool )
{
    // create time stamp
    var h = Date.now().getHours();
    var m = Date.now().getMinutes();
    var s = Date.now().getSeconds();
    var str = "\n" + h + ":" + m + ":" + s + " - " + val + "\n";
    // append highlight if bold = true
    if ( bold ) str = header() + str + footer();
    // ...
}
```

This should cover much of the basic data requirements. The majority of the code, here, simply outputs string values useful for configuring how your log data will look. If, for example, you want your log data to output to XML, it should be relatively trivial to override the header and footer functions to wrap the data in the required tags.

The next task you’re faced with is accepting a stream for output. Now, this class should be able to output to any stream, but at the same time, it needs to be able to create its own file stream by default. Therefore, the method for outputting data and the method for creating a stream should be separated, so that one might be overridden by the consuming application.

```javascript
public static function log( val : String, ?bold : Bool )
{
    // create time stamp
    var h = Date.now().getHours();
    var m = Date.now().getMinutes();
    var s = Date.now().getSeconds();
    var str = "\n" + h + ":" + m + ":" + s + " - " + val + "\n";
    // append highlight if bold = true
    if ( bold ) str = header() + str + footer();
```
Chapter 11: Performing Server-Side Trickery

// create stream
var stream = getStream();
// ...
}

public static function getStream() : neko.io.Output {
    var file = location;
    file += if ( !StringTools.endsWith( file, "\" ) &&
        !StringTools.endsWith( file, "/\" ) ) "/" else "";
    file += getLogFileName();
    return if ( neko.FileSystem.exists( file ) )
        neko.io.File.append( file, false )
    else
        neko.io.File.write( file, false );
}

All you’ve really done here is to provide a getStream method that returns a stream of type neko.io.Output. This should cater for any type of output stream, so you’re fully covered. If a particular application wants to provide its own stream type, it needn’t worry about parameter type confliction. This is why the method accepts no parameters, but instead relies entirely on static variables.

Now, you need to output the string data to the stream and then close it, remembering to first flush the stream of data:

public static function log( val : String, ?bold : Bool ) {
    // create time stamp
    var h = Date.now().getHours();
    var m = Date.now().getMinutes();
    var s = Date.now().getSeconds();
    var str = 
        "\n" + h + ":" + m + ":" + s + " - " + val + "\n";
    // append highlight if bold = true
    if ( bold ) str = header() + str + footer();
    // create stream
    var stream = getStream();
    // output bytes
    stream.writeBytes( str, 0, str.length );
    stream.flush();
    // close stream
    stream.close();
}

And that’s it. Your Logger class is now complete. All that’s left is to update the Exception class to cater for writes through the logger. Now, you might not always want to perform a log within an exception, especially if the output for the class is not Neko based. Therefore, a flag will need to be set to force the log entry. Another point worth considering is that you might not want all exceptions logged, so setting this flag as a static variable is probably not the best option. So what do you do?

Probably, the best course of action is to simply provide a log function that can be called — or not, whatever the case may be — when the exception is caught.
Part II: Server Side, JavaScript, and Flash: Oh My!

```java
public function log()
{
    #if neko
        Logger.log( this.toString(), true );
    #end
}
```

**Summary**

This chapter provided a good foundation for dealing with input and output in Neko applications. Specifically, you learned:

- Which database will best suit your type of Neko application
- How to access and modify data in a database
- How to use the SPOD framework
- How to work with files and streams
- How to create a logging class

In the next chapter, you look at building interactive client content for the Flash platform.
In this chapter you will discover the fundamental aspects of using haXe to produce Flash movies. The first section explains how to build and manage interactive contents; the second part highlights the differences between ActionScript and haXe and how the Flash API has been implemented.

The chapter covers in detail the following topics:

- What Flash is
- How to manage contents in Flash
- How to interact with Flash elements using the events
- How to use haXe from the point of view of an ActionScript developer
- How to use the Flash API in haXe

Many readers of this book have probably already created Flash movies using MTASC, the Adobe (formerly Macromedia) Flash IDE (last version is Flash CS3 Professional), or one of the Flex products (Flex Builder/Flex SDK). Those readers may find the first section of this chapter “The Flash Library” a little too basic for them and may want to jump directly to the “Flash to haXe” section.

The Flash Library

So what is Flash in the first place? Flash is an environment to define and reproduce rich multimedia contents. The original intent of the Flash developers was to integrate what was missing in the web environment: animations, sounds, and rich interactivity. At the time, programming dynamic HTML contents was actually very hard because of many incompatibilities between browsers and the very limited set of functionalities existent. Macromedia, now Adobe, has since created the Flash Player, a virtual machine capable of loading, parsing, and playing Flash movies.
Part II: Server Side, JavaScript, and Flash: Oh My!

It is not the purpose of this book to be an exhaustive guide and reference for the Flash API but it is designed to give the fundamentals to start working with Flash. If you want to dig deeper, you can find a lot of useful information on the Adobe website or in the many tutorials and guides available on the Internet and in the bookstores.

Adobe Flash has really a long history and it has evolved a lot in the past years. It was first introduced as an environment for designers to create animated graphics for the Web; progressively it has increased the possibilities for developers to control those graphics in a creative way. The language used by Flash has always been ActionScript (AS), a proprietary language very similar to the ECMAScript language. The first real appearance of AS in Flash version 5 was improved in Flash 6 and it was complemented by AS2 in Flash 7. At the time it was possible to choose to compile for Flash 6 or 7. The next version introduced new features mainly in a new class library that included the possibilities to upload files and manipulate bitmap data. Version 9, the current version, has been a very long step forward; AS is now at its version 3 and the underlying Virtual Machine (VM) completely revamped under the name of ActionScript Virtual Machine 2 (AVM 2). The old AVM 1 is still incorporated in the Flash Player to support legacy contents. The new VM is a lot more optimized for performances and the API library has been completely organized.

haXe provides a unique solution to target all the Flash versions still in use by now. With haXe you can compile your movies for Flash 6 to 9 and you can also create common code that can be compiled at once for more than one Flash version; when doing this you will find that there are consistent differences especially between Flash 6 to 8 and Flash 9. Those differences can be addressed using conditional compilation, as done several times in the examples in this book. If you plan to develop for a specific version of Flash, you don’t have really to worry about conditional compilation.

The Flash API for Flash version 6 to 8 is contained in the flash folder of the standard library, while for version 9 the folder is flash9; note however that the two define just one common package, flash with different implementations. The compiler picks up the correct one using the desired platform target.

The Flash Movie

A Flash movie is a compiled binary file with extension .swf; a single movie can contain multimedia resources (image, video, audio contents, and/or other Flash movies) and code blocks that are interpreted and executed by the Flash Player.

The Stage and the Timeline

Before introducing the Flash Library API it is important to understand the base mechanism that is behind a Flash movie. A movie is composed by a stage and a timeline. A stage is the display area where actors are placed on and where they actually act; actors can be pictures, drawings, or any kind of resource that has a visual representation. The timeline is a temporal segmentation: Each step in the segmentation can alter the properties of the objects on the stage. Using scripts it is possible to jump to a specific step, to stop the timeline execution, or to resume it. To give more tools to the developer, each stage can contain any number of substages, known as movie clips, that can be further nested and each with its own timeline. The movie clip expression is used generically in this chapter to indicate any object that has a visual representation. In Flash 9 this is not strictly correct because some of the classes that are displayable are not or do not extend the MovieClip class. Therefore, the expression is associated to some items in the speech more for their behavior than for their technical implementations.
The stage has a default irremovable root movie clip that contains every other movie clip in the scene. Movie clips are stacked in piles based on a depth-level value; the ones with higher depths are on top; at depth-level parity, the last movie clip defined goes on top. It is possible to swap depths at run time. Movie clips are layered, as represented in Figure 12-1.

The movie clip has by default a virtually unlimited canvas size; its actual size is determinate by the size and disposition of its contents; the display area can be limited using a mask. The movie clip coordinates are always relative to its container and in the case of the root movie clip, they will be relative to the containing player area. Coordinates are expressed in a Cartesian system of coordinates with the center in the top left corner, the positive x-axis on the right and the positive y-axis on the bottom, as illustrated in Figure 12-2.

**Flash Player**

The Flash Player is software by Adobe that permits the interaction with Flash movies. The software can be embedded as a plug-in for an existing software, like Firefox and Internet Explorer, or as a standalone player.
Part II: Server Side, JavaScript, and Flash: Oh My!

The Flash Player receives some parameters from the movie it is playing: the desired window size, the background color, and the frame rate. Some of these parameters and others can be specified when the player is embedded in an (X)HTML page. The following code is a classical template used to embed a movie in a page:

```html
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
    id="flashmovieid" width="640" height="480"
    codebase="http://fpdownload.macromedia.com/get/flashplayer/current/swflash.cab">
    <param name="movie" value="sample.swf" />
    <param name="quality" value="high" />
    <param name="bgcolor" value="#ffffff" />
    <param name="allowScriptAccess" value="sameDomain" />
    <embed src="sample.swf" quality="high" bgcolor="#ffffff" width="640" height="480" name="flashmovieid" align="middle" play="true" loop="false" quality="high" allowScriptAccess="sameDomain"
      type="application/x-shockwave-flash"
      pluginspage="http://www.adobe.com/go/getflashplayer">
    </embed>
</object>
```

The code uses the tag `object` to embed the player; a browser that does not recognize this element will try to use the included `embed`. This technique has many problems such as security issues, browser incompatibilities, invalid tags (`embed` is not a valid HTML element), and code repetition. A real valid
alternative solution seems to be using JavaScript to embed the player properly. Take a look at the following libraries that try to resolve this situation:

- **SWFObject**: A small JavaScript file used for embedding Adobe Flash content. The script can detect the Flash plug-in in all major web browsers (on Mac and PC) and is designed to make embedding Flash movies as easy as possible. It is also very search-engine friendly, degrades gracefully, can be used in valid HTML and XHTML 1.0 documents, and is forward compatible, so it should work for years to come (http://blog.deconcept.com/swfobject/).

- **Unobtrusive Flash Objects (UFO)**: A DOM script that detects the Flash plug-in and embeds Flash objects (files with the `.swf` extension). It has its roots in the Web Standards community and is designed to support W3C standards compliant, accessible and search engine friendly web design. It also contains several features and best practice techniques that other scripts currently don’t have (www.bobbyvandersluis.com/ufo/).

- **Flash Player Detection Kit**: Helps developers implement robust player detection for a variety of deployment environments by providing a set of templates and techniques to successfully detect the version of Flash Player installed on a user’s computer, and, if needed, to then install the latest version of Flash Player. The kit also includes a detailed explanation and sample files for implementing the new, player-based Flash Player Express Install experience (www.adobe.com/products/flashplayer/download/detection_kit/).

Descriptions have been taken directly from the project sites.

### The Movie Clip

After the brief introduction, it is time to pass to the action and coding some Flash routines. The first class to analyze is of course **MovieClip**. A movie clip has virtually all the possibilities to become a desired element on the stage; starting from a movie clip it is possible to implement simple user interface widgets, such as buttons and checkboxes, to display images and video and whatever.

### Creating a MovieClip in Flash 6 to 8

The **MovieClip** class is part of the **flash** package; it is not possible to directly create a **MovieClip** instance using the **new** keyword, instead you must use one of the following three methods from an already existing **MovieClip** instance: **createEmptyMovieClip()**, **duplicateMovieClip()**, or **attachMovie()**. In every Flash movie at least one **MovieClip** instance is always available that can be accessed using the static variable **flash.Lib.current**; this variable points to the root clip of the current movie. If a movie dynamically loads another one, the second also has its own root movie clip. The **flash.Lib.current** variable points to one or the other based on the context in which it is called. To refer to the very root **MovieClip** instance, use the variable **flash.Lib._root**.

In the following example, a square is drawn on a newly created movie clip; a second movie clip is added to the first with a smaller square inside. Ignore for the moment how the squares are generated and take a look at the **createMovieClip()** method. It relies on a passed **MovieClip** instance or on the **flash.Lib.current** if the argument is omitted, to generate a child **MovieClip**. The **createEmptyMovieClip()** used in the method, needs a string identifier for the object to create; the identifier can be used later on to reference the object by name. The second argument is a value for the depth level in the stack.
After the two boxes have been created, they are moved on the x-axis and rotated by 30 degrees. All the operations on the container movie clip, such as changing the coordinates, rotating and scaling, are applied to the container itself and to all of its children. The rotation and the scaling have the pivot point in the origin (point with coordinates 0, 0) of the actual movie clip.

import flash.Lib;
import flash.MovieClip;
class Main
{
    public static function main()
    {
        var mc = createMovieClip();
        drawRect(mc, 0, 0, 200, 200);
        var mc2 = createMovieClip(mc);
        drawRect(mc, 140, 140, 50, 50);
        mc._x = 110;
        mc._rotation = 30; // in degrees
    }
    #if flash6
    private static var nextid = 0;
    #end
    private static function createMovieClip(?parent : MovieClip) : MovieClip
    {
        if(parent == null)
            parent = Lib.current;
        #if flash6
        // flash 6 does not have the method getNextHighestDepth()
        // and the depth must be managed manually
        var id = ++nextid;
        #else true
        var id = parent.getNextHighestDepth();
        #end
        return parent.createEmptyMovieClip("mc_" + id, id);
    }
    private static function drawRect(mc:MovieClip,x:Float,y:Float,w:Float,h:Float)
    {
        mc.lineStyle(1, 0x000000);
        mc.moveTo(x, y);
        mc.lineTo(x+w, y);
        mc.lineTo(x+w, y+h);
        mc.lineTo(x, y+h);
        mc.lineTo(x, y);
    }
}

To compile the preceding example and the other examples in this section that target Flash 6 to 8, use the following command at the prompt/console:

```
> haxe -swf main8.swf -swf-version 8 -main Main --flash-strict -swf-header
280:280:21:ffffff
```
Chapter 12: Building Interactive Content with Flash

The resulting `sample.swf` can be opened in the standalone Flash Player or embedded in a web page. You can change the `-swf-version` to also target the Flash Player version 6 or 7 and you will obtain the same exact visual result.

The `duplicateMovieClip()` just creates a copy of the movie clip it is executed with; the `attachMovie()` will be described later in the “Extending the MovieClip class” section.

**Creating a MovieClip in Flash 9**

As you may have already discovered, the code in Flash 9 has been partitioned a lot; classes have been broken apart in a more organized way that fits better in a modern OOP environment. The `MovieClip` class is part of the `flash.display` package and inherits from a long chain of super classes.

The previous example has been rewritten to target the last Flash version currently available. In this version, you can instantiate a `MovieClip` object directly using the `new` keyword; the Movie Clip is automatically displayed as soon as it is appended to a visible container using the `addChild()` method as illustrated in the following example. You do not have to worry about depth levels because the `DisplayObjectContainer` class that is an ancestor of the MovieClip class handles them internally; this class also contains methods to swap and reposition the elements in the stack.

```actionscript
public class Main {
    public static function main() {
        var mc = createMovieClip();
        drawRect(mc, 0, 0, 200, 200);
        var mc2 = createMovieClip(mc);
        drawRect(mc, 140, 140, 50, 50);
        mc.x = 105;
        mc.rotation = 30; // in degrees
    }

    private static function createMovieClip(?parent : MovieClip) : MovieClip {
        if (parent == null) {
            parent = Lib.current;
            var mc = new MovieClip();
            parent.addChild(mc);
            return mc;
        }
    }

    private static function drawRect(mc:MovieClip,x:Float,y:Float,w:Float,h:Float) {
        var g = mc.graphics;
        g.lineStyle(1, 0x000000);
        g.drawRect(x, y, w, h);
    }
}
```
Part II: Server Side, JavaScript, and Flash: Oh My!

The preceding example and the others in this section that target Flash 9 can be compiled using the following command at the prompt/console:

```
> haxe -swf sample.swf -swf-version 9 -main Main --flash-strict -swf-
header 280:280:21:ffffff
```

**Text**

Text can be displayed in a Flash movie using the `TextField` class. The class can also be used to display an input box; to enable this feature, set the type field to `input` (in Flash 9 use the `TextFieldType.INPUT` to be sure to not misspell the value). For a read-only `TextField` use the default value `dynamic` (or `TextFieldType.DYNAMIC` in Flash 9).

### Creating a TextField in Flash 6 to 8

The `TextField` cannot be directly instantiated and can only be created starting from an existing `MovieClip` instance using the `createTextField()` method. The method accepts as arguments an instance name, a depth-level, two coordinates, and two dimensions for width and height.

```haxe
import flash.Lib;
import flash.MovieClip;
import flash.TextField;
class Main {
    public static function main()
    {
        #if flash8
        // in Flash 8, the createTextField() returns a reference
        // to the newly created object
        var t = Lib.current.createTextField("text", 0, 0, 0, 200, 20);
        #else flash
        // in previous version the create TextField must be referenced
        // by name
        Lib.current.createTextField("text", 0, 0, 0, 200, 20);
        var t : TextField = Reflect.field(Lib.current, "text");
        #end
        t.text = "Hello World!";
        t.type = "input";
        // alternatively you can set a text in HTML format
        // but first you must tell the TextField to treat it as such
        t.html = true;
        t.htmlText = "Hello <b>World!</b>";
    }
}
```

Starting at version 7 it is also possible to set a Cascading Style Sheet (CSS) using the property `TextField.styleSheet` and the HTML text can also contain references to embedded media like movie clips and images.
Chapter 12: Building Interactive Content with Flash

Creating a TextField in Flash 9

The TextField class is part of the flash.text package and it can be directly instantiated as it happens for the MovieClip class. As you can see from the next example, the code reorganization in Flash 9 has really improved the code usability and readability, and to perform the same actions as in previous versions, less typing is needed.

```actionscript
import flash.Lib;
import flash.display.MovieClip;
import flash.text.TextField;
import flash.text.TextFieldType;
class Main
{
    public static function main()
    {
        var t = new TextField();
        Lib.current.addChild(t);
        t.text = "Hello World!";
        t.type = TextFieldType.INPUT;
        // alternatively you can set a text in HTML format
        t.htmlText = "Hello <b>World!</b>";
    }
}
```

Multimedia Contents

There are two techniques to display multimedia contents inside a movie: embedding or dynamically loading. The first is explained in the next chapter in the “Assets with SWFMill” section dedicated to the SWFMill library. The latter uses a movie clip (a Loader in Flash 9) to load dynamically and to display contents.

Images and Movies

In the following examples, an image picture.jpg located in the same directory as the resulting .swf file will be loaded and immediately displayed. In a real-world application you will probably want to monitor the loading progress; this can be accomplished using the proper load events.

Remember that when you want to load a file from a relative URI as in the previous example, the path will be relative to the .swf position when the movie is played by the standalone Flash Player, but will be relative to the containing web page when the movie is embedded in an HTML document.

Remember also that the Flash security policy forbids loading remote files under certain circumstances.
Part II: Server Side, JavaScript, and Flash: Oh My!

Loading an Image in Flash 6 to 8

The `loadMovieMethod()` can load SWF, JPG, GIF (only unanimated), or PNG files. GIF, PNG, and progressive JPG have been introduced with version 8. To create the container movie clip, the same technique adopted earlier is applied.

```actionscript
import flash.Lib;
import flash.MovieClip;

class Main
{
    public static function main()
    {
        var mc = createMovieClip();
        mc.loadMovie("picture.jpg");
    }
}
```

```actionscript
#if flash6
private static var currentdepth = 0;
@end
private static function createMovieClip(?parent : MovieClip) : MovieClip
{
    if(parent == null)
        parent = Lib.current;
    #if flash6
    var depth = ++currentdepth;
    #else true
    var depth = parent.getNextHighestDepth();
    #end
    return parent.createEmptyMovieClip("mc_" + depth, depth);
}
```

Loading an Image in Flash 9

The `Loader` class, part of the `swf.display` package, can load SWF, JPG, GIF, or PNG files. Note that each time a request is created to reach a remote resource, the actual URI must be wrapped in an `URLRequest` object; this permits a finer grain control over the parameters of the request.

```actionscript
import flash.Lib;
import flash.display.Loader;
import flash.net.URLRequest;

class Main
{
    public static function main()
    {
        var loader = new Loader();
        loader.load(new URLRequest("picture.jpg"));
        Lib.current.addChild(loader);
    }
}
```
Chapter 12: Building Interactive Content with Flash

**Sounds**

Loading sounds is very easy in every Flash version; the supported sound format is the industry standard MP3.

**Loading a Sound in Flash 6 to 8**

The Sound class is in the flash package and it uses the loadSound() method to load a file. The sound stream starts as soon as the player begins to receive bytes if the second argument is set to `true`; otherwise it will not automatically play and the sound can only be reproduced after the complete load using the method `start()`. The `Sound` constructor accepts as an argument an instance of `MovieClip` or `Button`; when this argument is passed the `Sound` object is anchored to such an instance and it regulates only the sounds associated to it.

```ActionScript
import flash.Sound;
class Main{
    public static function main()
    {
        var s = new Sound(null);
        s.loadSound("sound.mp3", true);
    }
}
```

**Loading a Sound in Flash 9**

The Sound class is in the flash.media package. The `load()` method is used to load an audio file; alternatively it is possible to specify a file directly as an argument of the constructor. After the loading action, it is necessary to invoke the `play()` method to actually execute the file.

```ActionScript
import flash.media.Sound;
import flash.net.URLRequest;
class Main{
    public static function main()
    {
        var s = new Sound(new URLRequest("sound.mp3"));
        s.play();
    }
}
```

**Loading Values**

Loading text values from the server can be very useful and it is at the base of AJAX applications. The Flash environment offers a very simple way to communicate with the server using the `LoadVars` class in Flash 6 to 8 or the `URLLoader` class in Flash 9. Note that haXe comes with an integrated and feature-rich API to create communications between the different layers of an application; Chapter 14 is entirely dedicated to that.
Part II: Server Side, JavaScript, and Flash: Oh My!

**Loading Variables**

To test the Flash-to-server communication and to load a set of variables from it, a file `variables.txt` is created with the following contents:

```
message=Hello%20World&to=everyone
```

The content of the file must be encoded like the parameters of an HTTP GET request.

In the previous examples of loading images and sounds, there was no need to test the conclusion of the operation before using them; this is because they already have a place on the scene and they are automatically displayed or played as soon as they are available. However, variables cannot be used until they are fully loaded; there is no placeholder for the loading variables. For that reason, the following examples use a load event to monitor the availability of the requested data and to use them once they arrive. In a real-world application, it is probably desirable also to monitor possible communication errors and to check the loading status for long transactions.

**Loading Variables in Flash 6 to 8**

The `LoadVars` class is used to load values from a server. Once the `load()` method is executed and the data has been received, the loaded values will be available as new fields of the `LoadVars` instance.

```javascript
import flash.Lib;
import flash.LoadVars;

class Main {
    public static function main() {
        var loader = new LoadVars();
        loader.onLoad = function(success) {
            trace(loader.message);
            trace(loader.to);
        }
        loader.load("variables.txt");
    }
}
```

Alternatively you can use the `MovieClip.loadVariables()` method. Its behavior is very similar to the one described for the `LoadVars` class; the event to monitor to the data loading is `onData`. The method also accepts a second optional argument to designate the HTTP method, GET (default) or POST, used by the Flash Player to make the request to the server. Once loaded, the variables will be available as new fields of the invoking `MovieClip`.

**Loading Variables in Flash 9**

In Flash 9 the `URLLoader` loads the raw data and stores them as they are in the `data` variable. The `URLVariable` class is used to parse the received raw value and to transform it in a set of key-value pairs.
import flash.Lib;
import flash.net.URLLoader;
import flash.net.URLRequest;
import flash.net.URLVariables;
import flash.events.Event;
class Main
{
    public static function main()
    {
        var loader = new URLLoader();
        loader.addEventListener(Event.COMPLETE, function(event : Event)
        {
            var vars = new URLVariables(loader.data);
            trace(vars.message);
            trace(vars.to);
        });
        loader.load(new URLRequest("variables.txt"));
    }
}

Loading XML
The same techniques described previously to load variable values can be used to load an XML document. The response is treated as a whole and converted in an XML object using the Xml.parse() method.

Loading XML with Flash 6 to 8
To receive the result in its raw format, it is preferable to use the onData event that passes as an argument the unmodified result of the communication with the server:

import flash.Lib;
import flash.LoadVars;
class Main
{
    public static function main()
    {
        var loader = new LoadVars();
        loader.onData = function(src)
        {
            var xml = Xml.parse(src);
            trace(xml.toString());
        }
        loader.load("sample.xml");
    }
}
Part II: Server Side, JavaScript, and Flash: Oh My!

Loading XML with Flash 9

The URLLoader already provides the raw data as the response of the communication.

```ActionScript
import flash.Lib;
import flash.net.URLLoader;
import flash.net.URLRequest;
import flash.net.URLVariables;
import flash.events.Event;

class Main
{
    public static function main()
    {
        var loader = new URLLoader();
        loader.addEventListener(Event.COMPLETE, function(event : Event)
        {
            var xml = Xml.parse(loader.data);
            trace(xml.toString());
        });
        loader.load(new URLRequest("sample.xml"));
    }
}
```

Events

A simple event mechanism was introduced in the previous examples. An event is a special agreement between two entities: a dispatcher and an event handler. The dispatcher makes available a set of entry points at which an event handler can subscribe. The dispatcher will notify the handlers of the triggering of the subscribed events. To simplify the thing, imagine a hypothetical button object, the dispatcher, that has an `onClick` event (the `on` prefix is a widely used convention to identify the events easily). In the code, the developer can associate a function, the handler, to the `onClick`; the function is then executed each time the button is effectively clicked.

The Flash API provides a great number of events to cover the most common situations. The basics of events management is explained in the following sections; refer to the Flash documentation for further information.

Events in Flash 6 to 8

The base event model in Flash before version 9 is quite simple. The objects that need to expose events have functions starting with the `on` prefix. Those have by default an empty implementation; if they are invoked, they simply do nothing. The developer subscribes to the events simply providing a definition for such functions.

In the following example, a small green square is created with three event handlers; the first, `onMousePress`, is triggered when the mouse is pressed over the square; the second, `onRollOver`, and the third, `onRollOut`, are triggered respectively when the mouse enters the square area or leaves it.
import flash.Lib;
import flash.MovieClip;
class Main
{
    public static function main()
    {
        var mc = createMovieClip();
        drawRect(mc, 150, 10, 50, 50);
        mc.onPress = function()
        {
            trace("Clicked!");
        };
        mc.onRollOver = function()
        {
            mc._alpha = 50;
        };
        mc.onRollOut = function()
        {
            mc._alpha = 100;
        };
    }

    private static var currentdepth = 0;
    private static function createMovieClip(?parent : MovieClip) : MovieClip
    {
        if(parent == null)
            parent = Lib.current;
        #if flash6
        var depth = ++currentdepth;
        #else true
        var depth = parent.getNextHighestDepth();
        #end
        return parent.createEmptyMovieClip("mc_" + depth, depth);
    }

    // the function has been changed a little to fill the box with color
    private static function drawRect(mc:MovieClip,x:Float,y:Float,w:Float,h:Float)
    {
        mc.beginFill(0x00ff00);
        mc.moveTo(x, y);
        mc.lineTo(x+w, y);
        mc.lineTo(x+w, y+h);
        mc.lineTo(x, y+h);
        mc.lineTo(x, y);
        mc.endFill();
    }
}

The MovieClip also contains events for onMouseDown and onMouseUp, amongst others; note that these events are triggered every time the mouse button is used, even if it is not over the visible area of the MovieClip object.
Some events don’t have the usual signature `Void -> Void` (a function that has no arguments and returns nothing) but they pass a value. The `LoadVars.onLoad()` for example passes a Boolean argument stating if the loading communication has been successful or has failed for some reason.

The method as described earlier is not the only one to provide events; the `flash.MovieClipLoader` for example is a class used to monitor the loading progress of a target movie clip. The events are defined in `MovieClipLoader` instance but are effectively triggered by the target movie clip. Handlers in this case must be registered on the `MovieClipLoader` instance using the `addListener()` method. The `MovieClipLoader` was introduced in version 7.

**Events in Flash 9**

The event model in Flash 9 is quite a lot more sophisticated than in the previous versions. This sophistication opens possibilities harder to obtain otherwise, but it requires a little more attention. The dispatcher class, the class that provides events, has to implement the `flash.events.IEventDispatcher` interface. The `flash.events.EventDispatcher` class is a base class that implements the `IEventDispatcher` interface and that can be extended to provide the events features.

The `IEventDispatcher` interface defines methods that provide a fine control over events. It is possible to register more than one handler for each event with `addEventListener()`, it is possible to remove them using `removeEventListener()`, to check if an event is associated with a specific handler with `hasEventListener()`, or to invoke the handlers associated to the event with `dispatchEvent()`.

Differently from previous versions, there are no object fields associated to the events and they are referenced using a string identifier.

``` ActionScript 
import flash.Lib;
import flash.display.MovieClip;
import flash.events.MouseEvent;

class Main
{
    public static function main()
    {
        var mc = createMovieClip();
        drawRect(mc, 150, 10, 50, 50);

        mc.addEventListener(MouseEvent.CLICK, function(event)
        {
            trace("Clicked!");
        });

        mc.addEventListener(MouseEvent.MOUSE_OVER, function(event)
        {
            mc.alpha = 0.5;
        });

        mc.addEventListener(MouseEvent.MOUSE_OUT, function(event)
        {
            mc.alpha = 1;
        });
    }
}
```

``` ActionScript 
private static function createMovieClip(?parent : MovieClip) : MovieClip
{
    if(parent == null)
        parent = Lib.current;
```
The event handler function must accept a single argument of type flash.events.Event or a derived type and return nothing. The argument object contains additional information about the triggered event.

**Events over Time**

The Adobe Flash IDE has a visual editor that permits to design graphic elements defined as symbols and collected in a library that is embedded within the deployed movie. Each symbol has its own timeline and the designer can define frames in the timeline to alter the state of the drawing or to insert pieces of code to execute. The API provides many methods to deal with things such as jumping to a specific frame and stopping or playing the execution of a certain timeline, but it is not possible to create library symbols of this type programmatically. A new MovieClip instance, for example, has just one empty frame and it is not possible to add more at run time.

This does not mean that it is not possible to create visually rich elements at run time; it is just a matter of changing the point of view. To draw figures, you can rely on the drawing API that will be discussed later in this chapter. To manage the temporal events you can leverage on the enterFrame event to create transitions and time-dependant effects.

A more complex example follows to demonstrate the use of the enterFrame event and to resume what has been explained until now. A SlideShow class is created; its purpose is to load a set of images from the server and display them one after the other with a nice cross-fade effect. Images start to load at the class instantiation. The SlideShow.next() method displays the next image in the stack. This action is only possible after all the images have been preloaded; using it before simply does nothing. To use the next() method properly, the class has a single event (onLoad on Flash 6 to 8 and Event.COMPLETE on Flash 9) that can be used to monitor the loading status. In the Main.main() function a handler for the load event displays the first image as soon as all the collection is available; thereafter a new image fades in each time the user clicks the left mouse button somewhere on the movie.

The SlideShow constructor accepts an integer argument that is the total number of images to load; images must have a specific name (imgX.jpg where x is a number between 0 and the total number of images) and must be located in a subfolder called images. You can easily change the class to obtain the image paths from an array of values or better yet from a list generated by the server.

To obtain the best visual effect, use your image editor of choice to resize all the images to the same size as the Flash movie.
SlideShow with Flash 6 to 8

The `onEnterFrame` has been used two times in the example. The first is to monitor the load of each image. In the loop, it checks the byte size of the image; when all the bytes have been loaded the image is ready to be displayed and the loop is concluded. The same check could be done with the `MovieClipLoader` class, but this class is not available in Flash 6. The second time the `onEnterFrame` event has been used is inside the `easeOut` function that progressively changes the alpha value and the size of the fading-in image. The movement duration is expressed in seconds; to obtain a smoother effect, be sure to set a high frame rate for the movie.

```javascript
import flash.Lib;
import flash.MovieClip;
import flash.Stage;

class Main {
    public static function main()
    {
        var show = new SlideShow(8);
        show.onLoad = function()
        {
            show.next();
            Lib.current.onMouseDown = show.next;
        }
    }
}

class SlideShow {
    private static var PREFIX = "images/img";
    private static var EXTENSION = ".jpg";
    private var total : Int;
    private var pictures : Array<MovieClip>;
    private var current : Int;
    private var container : MovieClip;
    private var imagesLoaded : Bool;
    public function new(total : Int)
    {
        this.total = total;
        pictures = new Array();
        container = Lib.current.createEmptyMovieClip("container", 0);
        current = 0;
        imagesLoaded = false;
        loadNext();
    }
    private function loadNext()
    {
        if(pictures.length == total)
        {
            imagesLoaded = true; // no more pictures to load
            onLoad(); // trigger the onLoad event
            return;
        }
    }
```
Chapter 12: Building Interactive Content with Flash

var id = pictures.length;
var mc = container.createEmptyMovieClip("image" + id, id);
mc._visible = false;
pictures.push(mc);
load(mc, PREFIX+id+EXTENSION);

// exposed event
public function onLoad() {}  
private function load(target : MovieClip, path : String) {
    var me = this;
    container.onEnterFrame = function() {
        if(target.getBytesTotal() <= 0 ||
        target.getBytesLoaded() < target.getBytesTotal())
            return;
        target._visible = false;
        me.loaded(target);
    };
    target.loadMovie(path);
}
private function loaded(target : MovieClip) {
    container.onEnterFrame = null;
    loadNext();
}
public function next() {
    // prevent actions before all the images have been loaded
    if(!imagesLoaded)
        return;
    var mc = pictures[current];
    // the image that fades in must always be on top
    for(i in 0...total-1) {
        var ni = mc.getDepth()+1;
        if(ni >= total)
            ni -= total;
        mc.swapDepths(ni);
    }
    // set the initial values for the transition
    mc._alpha = 0;
    mc._visible = true;
    mc._xscale = 150;
    mc._yscale = 150;
    mc._x = (Stage.width - mc._width) / 2;
    mc._y = (Stage.height - mc._height) / 2;
    // set the transition with its duration in seconds
    easeOut(mc, 4);
    current = (current == total - 1) ? 0 : current + 1;
}

(continued)
(continued)

private static function easeOut(mc : MovieClip, time : Float)
{
    var properties = ["_alpha", "_xscale", "yscale", "_x", "_y"];
    var targetvalues = [100.0, 100.0, 100.0, 0, 0];
    var originals = new Array();
    var deltas = new Array();
    for(i in 0...properties.length)
    {
        originals.push(Reflect.field(mc, properties[i]));
        deltas.push(targetvalues[i]-Reflect.field(mc, properties[i]));
    }
    var start = Date.now().getTime();
    var step = 0.0;
    var duration = time * 1000;
    var tollerance = 0.1;
    mc.onEnterFrame = function()
    {
        step = Date.now().getTime() - start;
        if (step  <  duration)
        {
            for(i in 0...properties.length)
            {
                // cubic function to obtain an ease-out effect
                var x = step/duration - 1;
                var v = deltas[i]*(x*x*x+1)+originals[i];
                // adjust the value when in proximity of its target value;
                // avoids strange adjustments at the end of the transition
                if(v  <  targetvalues[i] + tollerance && v  >  targetvalues[i] - tollerance)
                {
                    v = targetvalues[i];
                    Reflect.setField(mc, properties[i], v);
                }
            }
        } else {
            mc.onEnterFrame = null;
        }
    };
}

To compile the preceding example, use the following command at the prompt/console:

> haxe -swf sample.swf -swf-version 8 -main Main --flash-strict -swf-
header 640:480:60:000000

The value of the switch -swf-version can be changed in 6 or 7.

**SlideShow with Flash 9**

As in the previous version the enterFrame event is used for the fade-in effect. The SlideShow class also extends the EventDispatcher to provide its Event.COMPLETE event.

```haxe
import flash.Lib;
import flash.display.DisplayObjectContainer;
import flash.display.Loader;
```
import flash.display.MovieClip;
import flash.display.Stage;
import flash.events.Event;
import flash.events.EventDispatcher;
import flash.events.MouseEvent;
import flash.net.URLRequest;
class Main {
    public static function main() {
        var show = new SlideShow(8);
        show.addEventListener(Event.COMPLETE, function(e) {
            show.next();
            Lib.current.addEventListener(MouseEvent.MOUSE_DOWN, function(e) {
                show.next();
            });
        });
    }
}
class SlideShow extends EventDispatcher {
    private static var PREFIX = "images/img";
    private static var EXTENSION = ".jpg";
    private var total : Int;
    private var pictures : Array<DisplayObjectContainer>;
    private var current : Int;
    private var container : MovieClip;
    private var imagesLoaded : Bool;
    public function new(total : Int) {
        super();
        this.total = total;
        pictures = new Array();
        container = new MovieClip();
        Lib.current.addChild(container);
        current = 0;
        imagesLoaded = false;
        loadNext();
    }
    private function loadNext() {
        var id = pictures.length;
        var loader = new Loader();
        loader.visible = false;
        loader.contentLoaderInfo.addEventListener(Event.COMPLETE, loaded);
        loader.load(new URLRequest(PREFIX+id+EXTENSION));
        container.addChild(loader);
        pictures.push(loader);
    }
}

(continued)
private function loaded(e : Event)
{
    if(pictures.length < total)
        loadNext();
    else {
        imagesLoaded = true;
        dispatchEvent(new Event(Event.COMPLETE)); // no more pictures to load
    }
}

public function next()
{
    if(!imagesLoaded)
        return;
    var mc = pictures[current];
    // the image that fades in must always be on top
    for(i in 0...total-1)
        container.swapChildrenAt(i, i+1);
    // set the initial values for the transition
    mc.alpha = 0.0;
    mc.visible = true;
    mc.scaleX = 1.5;
    mc.scaleY = 1.5;
    mc.x = (Lib.current.stage.stageWidth - mc.width) / 2;
    mc.y = (Lib.current.stage.stageHeight - mc.height) / 2;
    // set the transition with its duration in seconds
    easeOut(mc, 4);
    current = (current == total - 1) ? 0 : current + 1;
}

private static function easeOut(mc : DisplayObjectContainer, time : Float)
{
    var properties = ["alpha", "scaleX", "scaleY", "x", "y"];
    var targetvalues = [1.0, 1.0, 1.0, 0.0, 0.0];
    var originals = new Array();
    var deltas = new Array();
    for(i in 0...properties.length)
    {
        originals.push(Reflect.field(mc, properties[i]));
        deltas.push(targetvalues[i]-Reflect.field(mc, properties[i]));
    }
    var start = Date.now().getTime();
    var step = 0.0;
    var duration = time * 1000;
    var tollerance = 0.001;
    // in this case you cannot assign the function body directly to the variable,
    // otherwise its name will not available to be called recursively
    var frameHandler;
    frameHandler = function(e : Event)
Chapter 12: Building Interactive Content with Flash

```javascript
{  
  step = Date.now().getTime() - start;
  if (step < duration)
    {
      for(i in 0...properties.length)
        {
          // cubic function to obtain an ease-out effect
          var x = step/duration - 1;
          var v = deltas[i]*(x*x*x+1)+originals[i];
          // adjust the value when in proximity of its target value;
          // avoids strange adjustments at the end of the transition
          if(v < targetvalues[i] + tollerance && v > targetvalues[i] - tollerance)
            v = targetvalues[i];
          Reflect.setField(mc, properties[i], v);
        }
    } else
      mc.removeEventListener(Event.ENTER_FRAME, frameHandler);
  mc.addEventListener(Event.ENTER_FRAME, frameHandler);
}
```

To compile the preceding example, use the following command at the prompt/console:

```
> haxe -swf sample.swf -swf-version 9 -main Main --flash-strict -swf-header 640:480:60:000000
```

The Drawing API

The Flash drawing API enables you to generate vector graphics dynamically on top of visual objects (flash.MovieClip in Flash 6 to 8, and flash.display.Sprite in Flash 9). The API provides a way to draw lines and fill geometries with many different styles.

The biggest difference between the implementation of the drawing API in the last Flash version and the previous ones is that in the first case all the drawing methods have been collected in a `Graphics` class while in the latter case the methods are directly in the `MovieClip` class. The `graphics` property of the class `Sprite` is of type `Graphics`; `Sprite` is an ancestor to `MovieClip`. The Flash 9 implementation also has a richer set of drawing tools. The basic drawing has remained unchanged: First a starting point is determined using the method `moveTo()`, the default position is 0,0, then a style for the line is set with `lineStyle()` if not already defined, and then lines and curves are traced using respectively the `lineTo()` and `curveTo()` methods. To draw filled shapes, the defined path must be enclosed between the calls to the `beginFill()` and to the `endFill()` methods.

The following example demonstrates how very few lines of code leveraging the drawing API can be used to create beautiful drawings. The intent is to create a `Tree` class that generates random tree shapes. Some parameter may be adjusted and tuned to obtain very different results. Because the code to target
the various versions does not vary a lot, a unique code is written addressing the differences with the conditional compilation. The following code goes in a file named Tree.hx:

```haxe
#if flash9
import flash.display.Graphics;
#else flash
import flash.MovieClip;
#end

class Tree
{
    #if flash9
    public var container : Graphics;
    #else flash
    public var container : MovieClip;
    #end
    public var branches : Int;
    public var length : Float;
    public var angle : Float;
    public var width : Float;
    public var color : Int;
    public function new(c)
    {
        container = c;
        branches = 4;
        length = 80;
        angle = Angle.D90;
        width = 10;
        color = 0x000000;
    }
    public function generate(x:Float,y:Float,?len:Float,?ang:Float,?width:Float)
    {
        // when parameters are missing, generate the first branch using
        // the class parameters
        if(len == null)
            len = this.length;
        if(ang == null)
            ang = Angle.NORTH + Random.float(-this.angle/8, this.angle/8);
        if(width == null)
            width = this.width;
        // under those limits do not generate any more branches
        if(width < 1 || len < 2)
            return;
        // determine the coords of the end of the branch
        var nx = x+Math.cos(ang)*len;
        var ny = y+Math.sin(ang)*len;
        // draw the branch
        container.lineStyle(width, color, #if flash9 1 #else flash 100 #end);
        container.moveTo(x, y);
        container.lineTo(nx, ny);
        // generate the sub-branches
        for(i in 0...Random.int(1, branches))
    }
```

```haxe
#else flash
    public var container : MovieClip;
#endif
```
Chapter 12: Building Interactive Content with Flash

```actionscript

{ generate(
    nx,
    ny,
    Random.float(len*2/3, len),
    Random.float(-angle/2, angle/2)+ang,
    width-1);
}
}

class Random
{
    public static function float(min : Float, max : Float) : Float
    {
        return Math.random()*(max-min)+min;
    }
    public static function int(min : Int, max : Int) : Int
    {
        return Math.floor(float(min, max));
    }
}

class Angle
{
    public static var NORTH = -Math.PI/2;
    public static var D120  =  Math.PI*2/3;
    public static var D90   =  Math.PI/2;
    public static var D60   =  Math.PI/3;
    public static var D45   =  Math.PI/4;
    public static var D30   =  Math.PI/6;
    public static var D15   =  Math.PI/12;
}
```

There are just three lines in the whole code that make use of the drawing API and they are located in the method `Tree.generate()`; nevertheless the result can be quite complex.

The class in the preceding example can be used to generate one tree as follows:

```actionscript
import Tree;
import flash.Lib;
class Main
{
    // change these values to match the movie size
    private static var w = 640;
    private static var h = 480;
    public static function main()
    {
        #if flash9
        var tree = new Tree(Lib.current.graphics);
        #else flash
        var tree = new Tree(Lib.current);
        #end
        tree.generate(w/2, h);
    }
}
```
In Figure 12-3, the image of a tree is generated with the example code.

```
import Tree;
import flash.Lib;
#if flash9
import flash.display.GradientType;
import flash.display.MovieClip;
import flash.events.MouseEvent;
#endif
class Main
{
    // change these values to match the movie size
    private static var w = 640;
    private static var h = 480;
    public static function main()
```
Chapter 12: Building Interactive Content with Flash

{
#if flash9
  var paper = new MovieClip();
  var drawing = new MovieClip();
  Lib.current.addChild(paper);
  paper.addChild(drawing);
  papergfx = paper.graphics;
  drawinggfx = drawing.graphics;
  generateBackground();
  paper.addEventListener(MouseEvent.MOUSE_DOWN, function(e)
  {
    generateForest(e);
  });
#else flash
  var paper = Lib.current.createEmptyMovieClip("paper", 1);
  var drawing = paper.createEmptyMovieClip("drawing", 1);
  papergfx = paper;
  drawinggfx = drawing;
  generateBackground();
  paper.onMouseDown = generateForest;
#endif

#if flash9
private static var papergfx : flash.display.Graphics;
private static var drawinggfx : flash.display.Graphics;
#else flash
private static var papergfx : flash.MovieClip;
private static var drawinggfx : flash.MovieClip;
#endif

public static function generateForest()
{
  drawinggfx.clear();
  var tree = new Tree(drawinggfx);
  tree.branches = 4;
  tree.angle = Angle.D60;
  var trees = 30;
  var min = 0.6;
  var horizon = h*2/3;
  for(i in 0...trees) {
    var factor = ((i+1)/trees);
    var mfactor = factor*(1-min)+min;
    var rfactor = 1-mfactor;
    var r = 255-Math.floor(Random.float(40, 120)*factor);
    var g = 255-Math.floor(Random.float(40, 80)*factor);
    var b = 255-Math.floor(Random.float(40, 80)*factor);
    tree.color = (r << 16) | (g << 8) | b;
    tree.length = h/8*mfactor;
    tree.width = 10*mfactor;
    tree.generate(Random.float(0, w), horizon+(h-horizon)*factor*2/3);
  }
}

(continued)
public static function generateBackground()
{
    #if flash9
    var matrix = new flash.geom.Matrix();
    matrix.createGradientBox(w, h, -Math.PI/2, 0, 0);
    var gradient = GradientType.LINEAR;
    #else flash
    var matrix = { a:3e-17, b:-0.4, c:0.5, d:2e-17, tx:0, ty:h/2 };
    var gradient = "linear";
    #end
    papergfx.beginGradientFill(
        gradient,
        [0xdddddd, 0xffffff],
        [100, 100],
        [40, 100],
        matrix);
    papergfx.moveTo(0, 0);
    papergfx.lineTo(w, 0);
    papergfx.lineTo(w, h);
    papergfx.lineTo(0, h);
    papergfx.lineTo(0, 0);
    papergfx.endFill();
}  

This time many trees are produced at once; parameters have been changed on each generation. The
idea is to have brighter and shorter trees generated first and bigger trees on the front. The trees are also
disposed in a way that defines a visual perspective that is highlighted by the use of a gradient back-
ground. The background is drawn once at the movie loading. The forest is generated and refreshed on
each click of the mouse. The forest generation is a CPU intensive process and can require some time to
complete. Figure 12-4 shows an example of forest generated using the code provided previously.

**Extending the MovieClip Class**

The MovieClip class provides some solid bases to build upon, but sometimes, using only the features
that it provides can lead to repetitive code. Extending the MovieClip class can be a good mechanism to
improve the code efficiency. Creating a subclass in Flash 9 does not show any difficulty and it is done as
for every other class hierarchy; unfortunately, in Flash 6 to 8 it is not that easy. The problem is that the
MovieClip class cannot be directly instantiated and so do its derived classes.

To resolve the problem the MovieClip.attachMovie() method is used; it instantiates an object
from the library. The following example explains how to define a subclass MaskedClip from the base
MovieClip and instantiate it. The class adds a method setMaskSize() that creates a rectangular mask
for the current movie clip. A mask enables you to limit the visible area of an object. To define a mask on a
movie clip, a second one is created and associated to the first using the setMask() method. The shapes
drawn on the second movie clip are the visible area of the first.
import flash.Lib;
import flash.MovieClip;
class Main {
    public static function main() {
        var mc = cast(Lib.current.attachMovie("MaskedClip", "mymask", 0), MaskedClip);
        mc.setMaskSize(100, 100);
        mc.beginFill(0x00ff00);
        mc.moveTo(50, 50);
        mc.lineTo(120, 50);
        mc.lineTo(50, 120);
        mc.lineTo(50, 50);
        mc.endFill();
    }
}

(continued)
Part II: Server Side, JavaScript, and Flash: Oh My!

(continued)

```javascript
class MaskedClip extends MovieClip
{
    private var mask : MovieClip;
    public function setMaskSize(w : Float, h : Float)
    {
        if(mask == null) {
            mask = createEmptyMovieClip("mask", 0);
            setMask(mask);
        }
        mask.clear();
        mask.beginFill(0x000000);
        mask.moveTo(0, 0);
        mask.lineTo(w, 0);
        mask.lineTo(w, h);
        mask.lineTo(0, h);
        mask.lineTo(0, 0);
        mask.endFill();
    }
}
```

The example produces a diamond shape. Note that as the identifier for the symbol has been used, the same name of the class and how the movie clip was created with `attachMovie()` must be properly cast to be used as an instance of `MaskedClip`. Finally, even if it is possible to define a constructor that performs some initialization actions, it is not possible to define and pass arguments for it.

The same extending technique allows you to associate custom code to a symbol created with a visual editor such as the Flash IDE and embedded in the output movie using the `-swf-lib` switch. More on this in the next chapter, which discusses the use of haXe with the library SWFMill.

### Flash to haXe

ActionScript 2 is the Adobe (former Macromedia) syntax used to program Flash Movies from version 6 to 8, using the Flash IDE environment or MTASC by Nicolas Cannasse, the same author of haXe. The Flex 2 environment (the open source SDK or the commercial Flex Builder) makes use of ActionScript 3 to target Flash movies of version 9. haXe allows use of the same language to target any Flash version from 6 to 9 and also to generate AS3 files.

The haXe installation comes with a set of classes that map the original Flash API; because of the big differences and the incompatibilities between the Flash API for version 9 and the previous ones, two independent packages with the same `flash` name have been created.

In this section you will see the main differences in the syntax used in ActionScript (AS) and haXe. You’ll also see how the Flash API has been adapted to the haXe environment.

### Data Types

In AS2, variables are generic containers that can accept any kind of value; in AS3 and haXe, variables are strictly typed and so their values cannot be changed to another kind unless properly cast or converted; but they also provide a generic type that accepts any kind of value: `Object` for AS3 and `Dynamic` for haXe.

344
Another big difference between the three languages is that AS2 and haXe accept null values in place of the basic types, whereas AS3 will automatically convert them to their default values:

- 0 for Int (int in AS3) and UInt (uint)
- NaN for Float (Number in AS3)
- false for Bool (Boolean in AS3)

To overcome these differences haXe provides a Null<\text{T}> type that permits defining the aforementioned primitive types as null. In the vast majority of cases, replacing a value with its Null<\text{T}> counterpart has negligible performance effects, but for very intensive operations it is advisable to use only the basic types. Note also that a basic type used in place of an optional argument is implicitly transformed in its Null<\text{T}> counterpart.

Tables 12-1 and 12-2 describe some common AS2 and AS3 constructs and the equivalents in haXe. The tables also specify in which Flash versions the features are available.

**Table 12-1**

<table>
<thead>
<tr>
<th>ActionScript Type (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>* (F9)</td>
<td>Typed parameter or Dynamic</td>
</tr>
<tr>
<td>Boolean (F–69)</td>
<td>Bool</td>
</tr>
<tr>
<td>int (F9)</td>
<td>Int</td>
</tr>
<tr>
<td>Null (F9)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>Number (F6–9)</td>
<td>Float</td>
</tr>
<tr>
<td>Object (F6–9)</td>
<td>Dynamic</td>
</tr>
</tbody>
</table>

The * in AS3 denotes an untyped variable. In haXe the same can be obtained using type parameters like <\text{T}> or using Dynamic as a catchall solution.

The allowed values for Boolean are true and false. In haXe Bool is an Enum.

It is a 32-bit long signed integer. In Neko the Int size has been limited for performance reasons to 31-bit only. To use 32 bits also in Neko, use the neko.Int32 type.

Table continued on following page
Part II: Server Side, JavaScript, and Flash: Oh My!

<table>
<thead>
<tr>
<th>ActionScript Type (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>String (F6–9)</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>It is a data type to contain text values. It behaves almost the same on all platforms.</td>
</tr>
<tr>
<td>uint (F9)</td>
<td>UInt</td>
</tr>
<tr>
<td></td>
<td>It is a 32-bit unsigned integer. It is platform specific and its definition in haXe exists only when the target is Flash 9.</td>
</tr>
<tr>
<td>Void (F8)</td>
<td>Void</td>
</tr>
<tr>
<td></td>
<td>It is the type for functions that do not return anything. In haXe Void is an Enum with no constructors.</td>
</tr>
<tr>
<td>void (F9)</td>
<td></td>
</tr>
</tbody>
</table>

**Global Constants and Special Values**

Constant values and special values are globally accessible in both AS2/AS3. Their value is immutable and in haXe is generally coded as static variables of some classes, enums, or special keywords. Note that in haXe, constants cannot be defined.

**Table 12-2**

<table>
<thead>
<tr>
<th>ActionScript Value (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infinity (F6–9)</td>
<td>Math.POSITIVE_INFINITY</td>
</tr>
<tr>
<td></td>
<td>A special Number member expressing positive numeric infinity. In haXe its type is Float.</td>
</tr>
<tr>
<td>-Infinity (F6–9)</td>
<td>Math.NEGATIVE_INFINITY</td>
</tr>
<tr>
<td></td>
<td>A special Number member expressing negative numeric infinity. In haXe its type is Float.</td>
</tr>
<tr>
<td>NaN (F6–9)</td>
<td>Math.NaN</td>
</tr>
<tr>
<td></td>
<td>In AS it is a special member of the Number type that means that the value is not a number. In haXe the Math.NaN has the same role but obviously applies to the Float type.</td>
</tr>
<tr>
<td>undefined (F6–9)</td>
<td>null</td>
</tr>
<tr>
<td></td>
<td>The undefined value does not exist in haXe and is commonly replaced by the null value.</td>
</tr>
<tr>
<td>null (F6–9)</td>
<td>null</td>
</tr>
</tbody>
</table>

346
Top-level Functions

While AS makes an extended use of global functions, haXe has a very limited set of them and it is impossible for the developer to define new ones. The most notably top-level function is `trace`. Behind the scene the `trace` function is mapped to the `haxe.Log.trace()` static method, so in reality it is just a syntax shortcut.

In haXe the Flash top-level functions have been moved to some class definition. In an extreme case of necessity, when a flash functionality has not been introduced in haXe, it is possible to recur untyped blocks like the following:

```
// for Flash 6 to 8
untyped _global["System"]["capabilities"]
// for Flash 9
untyped __global__["flash.utils.getDefinitionByName"]
```

Nevertheless it is not advisable to recur to such construct unless absolutely necessary because syntax as in the preceding example is undocumented and subject to sudden changes in new versions.

In Table 12-3 you will notice that many AS functions have no correspondences in haXe; this is not because haXe developers are lazy, but because many of those functions are deprecated or not useful for developers in the current days. The Flash environment has been integrated in haXe with a lot of care and you will find everything you need in the current implementation.

Table 12-3

<table>
<thead>
<tr>
<th>ActionScript Function (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Array()</code> (F9)</td>
<td><code>new Array&lt;T&gt;()</code> or <code>[]</code></td>
</tr>
<tr>
<td></td>
<td>The <code>Array()</code> function in AS is a shortcut to create a new instance of an array. In haXe there is no equivalent function and arrays must be instantiated using the correct constructor or the square brackets shortcut.</td>
</tr>
<tr>
<td><code>true</code> (F6–9)</td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>false</code> (F6–9)</td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>newline</code> (F6–8)</td>
<td><code>\n</code></td>
</tr>
</tbody>
</table>

Table continued on following page
### ActionScript Function (Flash Version) vs. haXe Equivalent

<table>
<thead>
<tr>
<th>ActionScript Function (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean() (F9)</td>
<td>Std.bool()</td>
</tr>
<tr>
<td></td>
<td>Converts a value into a Boolean; the conversion is different from AS to haXe: in haXe <code>false</code>, <code>null</code>, and <code>0</code> are evaluated to <code>false</code>, all other values are evaluated to <code>true</code>.</td>
</tr>
<tr>
<td>clearInterval() (F6–8, F9 in flash.utils)</td>
<td>Use class haxe.Timer.</td>
</tr>
<tr>
<td></td>
<td>To perform an action similar to <code>clearInterval()</code> you have to instantiate a Timer object and call its <code>stop()</code> method when needed.</td>
</tr>
<tr>
<td></td>
<td><code>var t = new haxe.Timer();</code></td>
</tr>
<tr>
<td></td>
<td><code>t.run = function() {</code></td>
</tr>
<tr>
<td></td>
<td><code>    // do something</code></td>
</tr>
<tr>
<td></td>
<td><code>    }</code></td>
</tr>
<tr>
<td></td>
<td><code>    // do something else</code></td>
</tr>
<tr>
<td></td>
<td><code>t.stop();</code></td>
</tr>
<tr>
<td>clearTimeout() (F6–8, F9 in flash.utils)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>decodeURIComponent() (F9)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>decodeURI() (F9)</td>
<td>StringTools.urlDecode()</td>
</tr>
<tr>
<td></td>
<td>It also decodes whitespaces encoded as + symbols.</td>
</tr>
<tr>
<td>duplicateMovieClip() (F6–8)</td>
<td>Use the <code>duplicateMovieClip()</code> method of a <code>flash.MovieClip</code> instance.</td>
</tr>
<tr>
<td>encodeURI() (F9)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>encodeURIComponent() (F9)</td>
<td>StringTools.urlEncode()</td>
</tr>
<tr>
<td>escape() (F6–9)</td>
<td>StringTools.urlEncode()</td>
</tr>
<tr>
<td></td>
<td>Internally the method uses the Flash <code>escape()</code> function for versions 6–8 and <code>encodeURIComponent()</code> for Flash 9.</td>
</tr>
<tr>
<td>escapeMultiByte() (F9 in flash.utils)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>eval() (F6–8)</td>
<td>flash.Lib.eval()</td>
</tr>
<tr>
<td></td>
<td>The method accepts a string as an argument. When used in Flash version 6–8, the string can contain any valid AS2 expression to be evaluated; in Flash 9 the expression can only be a path to an object or to a field in the classic dot syntax.</td>
</tr>
<tr>
<td>fscommand() (F6–8, F9 in flash.system)</td>
<td>flash.Lib.fscommand()</td>
</tr>
</tbody>
</table>
## Chapter 12: Building Interactive Content with Flash

<table>
<thead>
<tr>
<th>ActionScript Function (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getClassByAlias()</code> (F9 in <code>flash.net</code>)</td>
<td>It is Action Message Format (AMF) specific. Similar functionalities can be obtained with the classes in the haxe.remoting package.</td>
</tr>
<tr>
<td><code>getDefinitionByName()</code> (F9 in <code>flash.utils</code>)</td>
<td><code>Type.resolveClass()</code></td>
</tr>
<tr>
<td><code>getInterval()</code> (F6–8, F9 in <code>flash.utils</code>)</td>
<td>See <code>clearInterval()</code></td>
</tr>
<tr>
<td><code>getProperty()</code> (F6–8)</td>
<td><code>Reflect.field()</code></td>
</tr>
<tr>
<td><code>getQualifiedClassName()</code> (F9 in <code>flash.utils</code>)</td>
<td><code>Type.getClassName()</code></td>
</tr>
<tr>
<td><code>getQualifiedSuperclass Name()</code> (F9 in <code>flash.utils</code>)</td>
<td>Use <code>Type.getSuperClass()</code> and then <code>Type.getClassName()</code></td>
</tr>
<tr>
<td><code>getTimer()</code> (F6–8, F9 in <code>flash.utils</code>)</td>
<td><code>flash.Lib.getTimer()</code></td>
</tr>
<tr>
<td><code>getURL()</code> (F6–8)</td>
<td><code>flash.Lib.getUrl()</code></td>
</tr>
<tr>
<td><code>getVersion()</code> (F6–8)</td>
<td><code>flash.Lib.getVersion()</code> or <code>flash.system.Capabilities.version</code></td>
</tr>
<tr>
<td><code>gotoAndPlay()</code> (F6–8)</td>
<td>Use the <code>gotoAndPlay()</code> method of a <code>flash.MovieClip</code> instance.</td>
</tr>
<tr>
<td><code>gotoAndStop()</code> (F6–8)</td>
<td>Use the <code>gotoAndStop()</code> method of a <code>flash.MovieClip</code> instance.</td>
</tr>
<tr>
<td><code>int()</code> (F9)</td>
<td><code>Std.int()</code></td>
</tr>
<tr>
<td><code>isFinite()</code> (F6–9)</td>
<td><code>Math.isFinite()</code></td>
</tr>
<tr>
<td><code>isNaN()</code> (F6–9)</td>
<td><code>Math.isNaN()</code></td>
</tr>
<tr>
<td><code>isXMLName()</code> (F9)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td><code>loadMovie()</code> (F6–8)</td>
<td>Use the <code>loadMovie()</code> method of a <code>flash.MovieClip</code> instance.</td>
</tr>
<tr>
<td><code>loadMovieNum()</code> (F6–8)</td>
<td>Not implemented.</td>
</tr>
</tbody>
</table>

*Table continued on following page*
### ActionScript Function (Flash Version) vs. haXe Equivalent

<table>
<thead>
<tr>
<th>ActionScript Function (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadVariables() (F6–8)</td>
<td>Use the loadVariables() method of a flash.MovieClip instance or the LoadVars class.</td>
</tr>
<tr>
<td>loadVariablesNum() (F6–8)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>MIMEExecute() (F7–8)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>navigateToURL() (F9 in flash.net)</td>
<td>flash.Lib.getUrl()</td>
</tr>
<tr>
<td>nextFrame() (F6–8)</td>
<td>Use the nextFrame() method of a flash.MovieClip instance.</td>
</tr>
<tr>
<td>nextScene() (F6–8)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>Number() (F6–9)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>Object() (F6–9)</td>
<td>In haXe a base Object type does not exist. You can obtain similar functionalities using the Dynamic type.</td>
</tr>
<tr>
<td>parseFloat() (F6–9)</td>
<td>Std.parseFloat()</td>
</tr>
<tr>
<td>parseInt() (F6–9)</td>
<td>Std.parseInt()</td>
</tr>
<tr>
<td>play() (F6–8)</td>
<td>Use the play() method of a flash.MovieClip instance.</td>
</tr>
<tr>
<td>prevFrame() (F6–8)</td>
<td>Use the prevFrame() method of a flash.MovieClip instance.</td>
</tr>
<tr>
<td>prevScene() (F6–8)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>print() (F6–8)</td>
<td>flash.Lib.print()</td>
</tr>
<tr>
<td>printAsBitmap() (F6–8)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>printAsBitmapNum() (F6–8)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>printNum() (F6–8)</td>
<td>Not implemented.</td>
</tr>
</tbody>
</table>
### Chapter 12: Building Interactive Content with Flash

<table>
<thead>
<tr>
<th>ActionScript Function (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>registerClassAlias() (F9 in flash.net)</td>
<td>It is Action Message Format (AMF) specific. Similar functionalities can be obtained with the classes in the haxe.remoting package.</td>
</tr>
<tr>
<td>removeMovieClip() (F6–8)</td>
<td>Use the removeMovieClip() method of a flash.MovieClip instance.</td>
</tr>
<tr>
<td>sendToUrl() (F9 in flash.net)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>setInterval() (F6–8, F9 in flash.utils)</td>
<td>See clearInterval().</td>
</tr>
<tr>
<td>setProperty() (F6–8)</td>
<td>Reflect.setField()</td>
</tr>
<tr>
<td>setTimeout() (F6–8, F9 in flash.utils)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>showRedrawRegions() (F8, F9 in flash.profiler)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>startDrag() (F6–8)</td>
<td>Use the startDrag() method of a flash.MovieClip instance.</td>
</tr>
<tr>
<td>stop() (F6–8)</td>
<td>Use the stop() method of a flash.MovieClip instance.</td>
</tr>
<tr>
<td>stopAllSound() (F6–8)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>stopDrag() (F6–8)</td>
<td>Use the stopDrag() method of a flash.MovieClip instance.</td>
</tr>
<tr>
<td>String() (F6–9)</td>
<td>Std.string()</td>
</tr>
<tr>
<td>targetPath() (F6–8)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>trace() (F6–9)</td>
<td>trace()</td>
</tr>
<tr>
<td>uint() (F9)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>unescape() (F6–9)</td>
<td>StringTools.urlDecode() Internally the method uses the Flash escape() function for versions 6–8 and encodeURIComponent() for Flash 9.</td>
</tr>
<tr>
<td>unescapeMultiByte() (F9 in flash.utils)</td>
<td>Not implemented.</td>
</tr>
</tbody>
</table>

Table continued on following page
Part II: Server Side, JavaScript, and Flash: Oh My!

Operators

The vast majority of operators in AS have been mapped equally in haXe. The most notable exceptions or the one that needs some further descriptions are described in Table 12-4; others of less common use have been omitted.

Table 12-4

<table>
<thead>
<tr>
<th>ActionScript Operator (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array access operator: [] (F6–9)</td>
<td>The array access operator works the same for the three forms of syntax for creating and accessing array elements. In haXe, because arrays are strongly typed and no type parameter can be expressed with this syntax, the type of the array is inferred by the most accommodating type for the elements existing at the moment of the declaration.</td>
</tr>
<tr>
<td>as (F9)</td>
<td>There is no direct equivalent but you can obtain the same effect with something like the following:</td>
</tr>
<tr>
<td></td>
<td>var b; try { b = cast(a, B); } catch(e :Dynamic){ b = null; }</td>
</tr>
</tbody>
</table>
## Chapter 12: Building Interactive Content with Flash

<table>
<thead>
<tr>
<th>ActionScript Operator (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>delete</strong> (F6–9)</td>
<td>Reflect.deleteField()</td>
</tr>
<tr>
<td>Field access operator: [] (F6–9)</td>
<td>Reflect.field() to read. Reflect.setField() to change the value. To iterate over object fields use Reflect.fields(); class methods will be ignored. Reflect.setField() cannot be used to redefine a method in Flash 9 unless it is modified with the keyword <code>f9dynamic</code>.</td>
</tr>
<tr>
<td><strong>instanceof</strong> (F6–9)</td>
<td>Std.is()</td>
</tr>
<tr>
<td><strong>is</strong> (F9)</td>
<td>Std.is()</td>
</tr>
<tr>
<td><strong>Name qualifier:</strong> :: (F9)</td>
<td>Not implemented in haXe.</td>
</tr>
<tr>
<td><strong>Object initializer:</strong> {} (F6–9)</td>
<td>( /* field definitions */ ) or Reflect.empty()</td>
</tr>
<tr>
<td><strong>Regular expression:</strong> /pattern/flag (F9)</td>
<td>The ActionScript regular expression operator instantiates a new RegExp object while the haXe operator instantiates an ERegExp object. They are functionally equivalent.</td>
</tr>
<tr>
<td><strong>Type operator:</strong> : (F6–9)</td>
<td>The operator works in the same way in ActionScript and haXe; however declaring types in haXe is in many cases superfluous because the compiler correctly infers types from the context.</td>
</tr>
<tr>
<td><strong>typeof</strong> (F6–9)</td>
<td>Std.typeof()</td>
</tr>
<tr>
<td><strong>XML operators</strong> (F9)</td>
<td>Not implemented in haXe.</td>
</tr>
</tbody>
</table>
Statements, Structures, and Keywords

There are many affinities between the general syntax of ActionScript and haXe. Table 12-5 explains the subtle differences and highlights the entries that do not have an equivalent implementation.

Table 12-5

<table>
<thead>
<tr>
<th>ActionScript Keyword (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>arguments (F6–9)</td>
<td>haXe does not have a unified way to access to the arguments of a function as an array of values. In the majority of the cases the problem can be overcome using the Reflect.makeVarArgs() as a workaround. Otherwise it is possible to use an untyped expression such as untyped <strong>arguments</strong> that is valid for Flash from version 6–8.</td>
</tr>
<tr>
<td>break (F6–9)</td>
<td>break works in loop structures (for, while, do…while) but not inside a switchcase. See switch.</td>
</tr>
<tr>
<td>class (F6–9)</td>
<td>Defining a class in AS and haXe is almost identical. The biggest and more evident difference is that AS uses the convention to name the class constructor after the class name while in haXe the constructor of any class always has the name new.</td>
</tr>
<tr>
<td>switch (F6–9)</td>
<td>Each case in a switch statement is a separate block of instructions and it is not possible to fall through from one to another. Seen from an ActionScript point of view, it is as if all the cases are automatically ended with a break statement.</td>
</tr>
<tr>
<td>dynamic (F6–9)</td>
<td>class Sample implements Dynamic {}</td>
</tr>
<tr>
<td>for (F6–9)</td>
<td>haXe only supports the for syntax in one of the two possible flavors: for(i in 0…size) { /<em>…</em>/ } Where size is an integer value and i is an integer local variable that is automatically incremented on each loop. for(i in iter) { /<em>…</em>/ } Where iter is an Iterable or Iterator object and i is a local variable that assumes the value of the current element in the sequence. Note that there is no need to use the var statement and assign a type for the variable I in both cases; the variable is automatically generated and the type is inferred in the loop.</td>
</tr>
<tr>
<td>for..in (F6–9)</td>
<td>See for and the Reflect description in Chapter 16.</td>
</tr>
<tr>
<td>for each..in (F9)</td>
<td>See for and the Reflect description in Chapter 16.</td>
</tr>
<tr>
<td>implements (F6–9)</td>
<td>In haXe, a class can implement not only interfaces but also classes. This feature is not supported in Flash 9.</td>
</tr>
</tbody>
</table>
### Chapter 12: Building Interactive Content with Flash

<table>
<thead>
<tr>
<th>ActionScript Keyword (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>intrinsic (F6–8)</td>
<td>Intrinsic in AS2 and native in AS3 have a meaning very similar to <code>extern</code> in haXe. Take a look at Chapter 6 to see how to use the <code>extern</code> class modifier.</td>
</tr>
<tr>
<td>native (F9)</td>
<td></td>
</tr>
<tr>
<td>private (F6–9)</td>
<td>In ActionScript 3, <code>private</code> fields are not accessible in derived classes but <code>protected</code> fields are. In haXe, only <code>private</code> fields are visible in the declaring class and in the derived classes as it happens in ActionScript 2.</td>
</tr>
<tr>
<td>get/set (F6–9)</td>
<td>Variable getters and setters in haXe are implemented using the variable access modifiers. See Chapter 5 for further instructions.</td>
</tr>
<tr>
<td>set variable F(6–8)</td>
<td>This statement does not make any sense in haXe because a global variable does not exist. Use <code>Reflect.setField()</code> to set object variables.</td>
</tr>
<tr>
<td>try..catch..finally (F7–9)</td>
<td>In haXe there is no <code>finally</code> statement and a type for the captured error must always be specified. The catch all type is <code>Dynamic</code>. In Flash 9 the <code>try..catch</code> block is not allowed inside an expression.</td>
</tr>
<tr>
<td>with (F6–9)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>label (F9)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>final (F9)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>import (F6–9)</td>
<td>When a type is imported in ActionScript but not used, the resulting swf will not include that portion of code; in haXe, an import implies the inclusion of the code even if the imported types are not used. This is important because you may wish to run code that is invoked indirectly using reflection and this code must be present to be accessed. Using imports only when they are needed is very important to keep the compiled output as lean as possible.</td>
</tr>
<tr>
<td>include (F9)</td>
<td>Not implemented; see import.</td>
</tr>
<tr>
<td>internal (F9)</td>
<td>Internal is not implemented in haXe but <code>private</code> types (<code>classes</code>, <code>enums</code>, and/or <code>typedefs</code>) have a very similar role. A <code>private</code> type is only accessible in the file in which it is defined.</td>
</tr>
<tr>
<td>protected (F9)</td>
<td>See <code>private</code>.</td>
</tr>
<tr>
<td>Rest arguments: ... (F9)</td>
<td>Not implemented.</td>
</tr>
</tbody>
</table>

*Table continued on following page*
Part II: Server Side, JavaScript, and Flash: Oh My!

<table>
<thead>
<tr>
<th>ActionScript Keyword (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>const (F9)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>namespace (F9)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>package (F9)</td>
<td>Packages have the same role in AS3 as in haXe but with a different syntax. In AS3, package contents are included in curly brackets while in haXe a package is a single line statement valid for the entire file where it is declared.</td>
</tr>
<tr>
<td>use namespace (F9)</td>
<td>Not implemented.</td>
</tr>
</tbody>
</table>

**Global Properties**

haXe does not support global properties as a language feature, the ones existing from the Flash API have been mapped to some class definition or omitted (see Table 12-6).

**Table 12-6**

<table>
<thead>
<tr>
<th>ActionScript Property (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>_accProps (F6.0.65–8)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>_focusrect (F6–8)</td>
<td>Use the _focusrect variable of flash.MovieClip, flash.Button, or flash.TextField.</td>
</tr>
<tr>
<td>_global (F6–8)</td>
<td>flash.Lib._global</td>
</tr>
<tr>
<td>_level (F6–8)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>_parent (F6–8)</td>
<td>Use the _parent variable of flash.MovieClip, flash.Button, or flash.TextField.</td>
</tr>
<tr>
<td>_quality (F6–8)</td>
<td>Use the _quality variable of flash.MovieClip, flash.Button, or flash.TextField.</td>
</tr>
<tr>
<td>_root (F6–8)</td>
<td>flash.Lib._root</td>
</tr>
<tr>
<td>_soundbuftime (F6–8)</td>
<td>Use the _soundbuftime variable of flash.MovieClip, flash.Button, or flash.TextField.</td>
</tr>
</tbody>
</table>
Chapter 12: Building Interactive Content with Flash

*Top-level Classes Replaced by haXe-Specific Types*

The top-level classes of the Flash API are located in the `flash` package. Some of the top-level classes do not have an equivalent class in haXe. Table 12-7 explains how to work around the absence.

**Table 12-7**

<table>
<thead>
<tr>
<th>ActionScript Class (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array (F6–9)</td>
<td>Array&lt;T&gt;</td>
</tr>
<tr>
<td>Arrays and collections in general are strictly typed in haXe. The type parameter T is used to express the correct type. //&quot;a&quot; is an empty array of integer values.</td>
<td></td>
</tr>
<tr>
<td>var a = new Array&lt;Int&gt;();</td>
<td></td>
</tr>
<tr>
<td>The majority of the fields that exist in the AS Array classes have a corresponding field in the haXe Array class; some methods have been moved in the class Lambda that introduces much functionality to deal with series.</td>
<td></td>
</tr>
<tr>
<td>AsBroadcaster (F6–8)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>Boolean (F6–9)</td>
<td>Bool</td>
</tr>
<tr>
<td>In haXe, a Boolean value is not an instance of a class but the constructor of an enumerator. In AS the Boolean instance has a method <code>toString()</code> to print true or false; the same can be obtained in haXe using the function <code>Std.string(boolvalue)</code>.</td>
<td></td>
</tr>
<tr>
<td>Class (F9)</td>
<td>Class</td>
</tr>
<tr>
<td>To get the class of an object use <code>Type.getClass()</code>; to instantiate an object from a Class definition use <code>Type.createInstance()</code>.</td>
<td></td>
</tr>
<tr>
<td>CustomActions (F6–8)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>Date (F6–9)</td>
<td>haXe has adopted a unified Date class that behaves the same on all the target platforms, not just Flash, and differs a little from the Flash implementation. Some methods found in the Flash Date class have been moved to the DateTools class.</td>
</tr>
<tr>
<td>Error (F6–9)</td>
<td>haXe permits to throw and catch any type of variable so it is not needed to implement Error for error handling.</td>
</tr>
</tbody>
</table>

*Table continued on following page*
Part II: Server Side, JavaScript, and Flash: Oh My!

<table>
<thead>
<tr>
<th>ActionScript Class (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
</table>
| Function (F6–9)                   | Functions are not class instances in haXe; nevertheless it is possible to invoke dynamically a function using `Reflect.callMethod()` or it is possible to reassign a function body to an already existing definition.  
```haXe
class Sample
{
    public static f9dynamic function hello()
    {
        return "hi";
    }
    public static function main()
    {
        trace(Sample.hello());
        Sample.hello = function()
        {
            return "Hello World!";
        }
        trace(Sample.hello());
    }
}
```
| haXe supports method redefinition on all the platforms but on Flash 9 the methods to redefine must be marked with the keyword `f9dynamic`. This has no effect on the others platforms. To allow this behavior, `f9dynamic` methods are compiled as variables of type `Function`; a limitation of this implementation is that a function variable cannot have direct access to the same method defined in a parent class. |
| int (F9)                          | Int  
The `Int` type in haXe has no fields. To manipulate an `Int` value, use the methods from `Std, Math`, and other classes. |
| Math (F6–9)                       | Math  
Practically every static field that exists in the `Math` class of the ActionScript language has an equivalent in haXe in its `Math` class. The following constant values have no equivalent yet: `E, LN10, LN2, LOG10E, LOG2E, SQRT1_2, SQRT2`. |
| Namespace (F9)                    | Not implemented. |
| Number (F6–9)                     | Float  
The `Float` type in haXe has no fields. To manipulate a `Float` value use the methods from `Std, Math` and other classes. |
Chapter 12: Building Interactive Content with Flash

<table>
<thead>
<tr>
<th>ActionScript Class (Flash Version)</th>
<th>haXe Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object (F6–9)</td>
<td>Use the haXe Dynamic type to provide similar functionalities as the ActionScript Object. Unwatch(), watch() and addProperty() are not available in haXe</td>
</tr>
<tr>
<td>QName (F9)</td>
<td>Not implemented.</td>
</tr>
<tr>
<td>RegExp (F9)</td>
<td>EReg</td>
</tr>
<tr>
<td>String (F6–9)</td>
<td>String</td>
</tr>
<tr>
<td>uint (F9)</td>
<td>UInt</td>
</tr>
<tr>
<td>XML (F9)</td>
<td>Xml</td>
</tr>
<tr>
<td>XMLList (F9)</td>
<td>Not implemented, see XML.</td>
</tr>
<tr>
<td>XMLNode (F6–8)</td>
<td>Not implemented, see XML.</td>
</tr>
<tr>
<td>XMLUI (F7–8)</td>
<td>Not implemented, see XML.</td>
</tr>
</tbody>
</table>

**Flash API Classes**

haXe enforces the convention to use a capital letter as the first character of a class name returning a compilation error otherwise; for this reason the two classes capabilities and security (in Flash 6 to 8) have been renamed Capabilities and Security.

The haXe implementation also provides a special class flash.Lib that has several useful methods and variables whose purposes have been already explained in the previous tables. The class does not map to any class in the Flash API.

**Version 6 to 8**

Table 12-8 lists all the classes that have been implemented in haXe and the flash version of their adoption. Note otherwise that some changes have occurred in the method signatures of some classes; refer to the official Flash documentation for further indications.
Part II: Server Side, JavaScript, and Flash: Oh My!

Table 12-8

<table>
<thead>
<tr>
<th>Package</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>flash</td>
<td>Accessibility (F6–8), Button (F6–7, added features in F8), Camera (F6–8), Color (F6–8),</td>
</tr>
<tr>
<td></td>
<td>ContextMenu (F7–8), ContextMenuItem (F7–8), Key (F6–8), LoadVars (F6–7, added features in F8),</td>
</tr>
<tr>
<td></td>
<td>Local, LocalConnection (F6–8), Microphone (F6–8), Mouse (F6–8), MovieClip (F6, added features in F7 and F8), MovieClipLoader (F7–8), NetConnection (F7–8), NetStream (F7, added features in F8), PrintJob (F7–8), Selection (F6–8), SharedObject (F6–8), Sound (F6–8), Stage (F6–8), System (F6–8), TextField (F6–7, added features in F8), TextFormat (F6–7, added features in F8), TextSnapshot (F7–8), Video (F6, added features in F7 and F8), XMLSocket (F6–8)</td>
</tr>
<tr>
<td>flash.display</td>
<td>BitmapData (F8)</td>
</tr>
<tr>
<td>flash.external</td>
<td>ExternalInterface (F8)</td>
</tr>
<tr>
<td>flash.filters</td>
<td>BevelFilter (F8), BitmapFilter (F8), BlurFilter (F8), ColorMatrixFilter (F8), ConvolutionFilter (F8), DisplacementMapFilter (F8), DropShadowFilter (F8), GlowFilter (F8), GradientBevelFilter (F8), GradientGlowFilter (F8)</td>
</tr>
<tr>
<td>flash.geom</td>
<td>ColorTransform (F8), Matrix (F8), Rectangle (F8), Point (F8), Transform (F8)</td>
</tr>
<tr>
<td>flash.net</td>
<td>FileReference (F8), FileReferenceList (F8)</td>
</tr>
<tr>
<td>flash.system</td>
<td>Capabilities (F6–8), IME (F8), Security (F6–7, added features in F8)</td>
</tr>
<tr>
<td>flash.text</td>
<td>StyleSheet (F7–8), TextRenderer (F8)</td>
</tr>
</tbody>
</table>

Version 9

Table 12-9 lists all the classes that are implemented in haXe. Other types, such as enums have been omitted. The definitions for Flex library are available in the haxelib project flex.

Table 12-9

<table>
<thead>
<tr>
<th>Package</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>flash</td>
<td>Event</td>
</tr>
<tr>
<td>flash.accessibility</td>
<td>Accessibility, AccessibilityProperties</td>
</tr>
<tr>
<td>flash.display</td>
<td>AVM1Movie, Bitmap, BitmapData, DisplayObject, DisplayObjectContainer,</td>
</tr>
<tr>
<td></td>
<td>FrameLabel, Graphics, InteractiveObject, Loader, LoaderInfo, MorphShape,</td>
</tr>
<tr>
<td></td>
<td>MovieClip, Scene, Shape, SimpleButton, Sprite, Stage</td>
</tr>
</tbody>
</table>
## Chapter 12: Building Interactive Content with Flash

<table>
<thead>
<tr>
<th>Package</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>flash.external</td>
<td>ExternalInterface</td>
</tr>
<tr>
<td>flash.filters</td>
<td>BevelFilter, BitmapFilter, BitmapFilterQuality, BitmapFilterType, BlurFilter, ColorMatrixFilter, ConvolutionFilter, DisplacementMapFilter, DisplacementMapFilterMode, DropShadowFilter, GlowFilter, GradientBevelFilter, GradientGlowFilter</td>
</tr>
<tr>
<td>flash.geom</td>
<td>ColorTransform, Matrix, Point, Rectangle, Transform</td>
</tr>
<tr>
<td>flash.media</td>
<td>Camera, ID3Info, Microphone, Sound, SoundChannel, SoundLoaderContext, SoundMixer, SoundTransform, Video</td>
</tr>
<tr>
<td>flash.printing</td>
<td>PrintJob, PrintJobOptions, PrintJobOrientation</td>
</tr>
<tr>
<td>flash.text</td>
<td>AntiAliasType, CSMSettings, Font, FontStyle, FontType, GridFitType, StaticText, StyleSheet, TextColorType, TextDisplayMode, TextExtent, TextField, TextFieldAutoSize, TextFieldStyle, TextFormat, TextFormatAlign, TextFormatDisplay, TextStyle, TextLineMetrics, TextRenderer, TextRun, TextSnapshot</td>
</tr>
<tr>
<td>flash.ui</td>
<td>ContextMenu, ContextMenuBuiltInItems, ContextMenuItem, Keyboard, KeyLocation, Mouse</td>
</tr>
<tr>
<td>flash.utils</td>
<td>ByteArray, Dictionary, Endian, ObjectInput, ObjectOutput, Proxy, SetIntervalTimer, Timer</td>
</tr>
<tr>
<td>flash.xml</td>
<td>XMLDocument, XMLNode, XMLNodeType, XMLElement, XMLParser, XMLTag</td>
</tr>
</tbody>
</table>
Part II: Server Side, JavaScript, and Flash: Oh My!

Summary

It has been a long journey in the world of Flash interactivity. In this chapter you collected the basics and more to build powerful and richly interactive movies. The following topics were discussed:

- What elements compose a Flash Movie
- How to create a MovieClip and a TextField
- How to load, display or play images, sounds, and contents from a server
- What the events are and how they are managed to create interactions with the movie
- What the differences are between ActionScript and haXe
- How the Flash API has been implemented in haXe

The next chapter discusses how to manage assets with SWMill and how to create a simple user interface control.
An integrated development environment (IDE) may be a very important component in the working environment of a programmer. Adobe actually distributes the IDE to target Flash, the Flash CS3 Professional software that is aimed principally at designers, and the Adobe Flex Builder 2 that targets software developers. haXe is still young and does not have a lot of editors yet although new solutions are always emerging, but the very important fact is that haXe does not necessarily need a complex IDE to be fully functional and usable. Debugging in haXe is quite easy because the compiler is very smart in detecting inconsistencies and errors.

One of the biggest advantages of the Adobe Flash CS3 product is the presence of a visual editor that allows creating and importing graphics that can be embedded as symbols in the output movie and used by the code. Because haXe is a programming language and not an IDE, such functionalities must be provided using external tools. It is of course possible to use the Adobe products to integrate the haXe environment, but it is also possible to rely on a plethora of open source editors that can create and manipulate digital images, vector drawing, sounds, and videos. The problem is not how to produce the contents, but how to use and to integrate them with haXe.

In this chapter, the following topics will be discussed:

- What SWFMill is
- How to embed images and movies
- How to embed fonts
- How to define a user interface
- How to create a user interface library in haXe
Assets with SWFMill

SWFMill is a command-line tool created by Daniel Fischer, contributed by many developers and distributed under the open source GNU General Public License (GPL). Although the development is continuous, the author does not consider the code mature enough to release a version “one” of the tool. Nevertheless, the library can be confidently used to provide and integrate functionalities in many areas. The main purpose of the tool is to produce and interpret SWF files. Starting from an XML definition following the swfml format, it is possible to generate an SWF movie; and from an SWF movie it is possible to generate an XML definition. The Flash format is not completely covered, and there are still holes in Flash 8 and particularly Flash 9 areas. The tool needs virtually no installation and just has a bunch of dependencies that comes embedded for the lucky Windows users. You can find more information on the usage and installation on the official website at http://swfmill.org/. The last version at the moment of writing is 0.2.12.

The SWFMill command (swfmill) accepts the parameters described in Table 13-1; parameters between square brackets are optional.

Table 13-1

<table>
<thead>
<tr>
<th>SWFMill Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml2swf &lt;input&gt;</td>
<td>Transforms the input (an XML file or a string from stdin) in an SWF file and writes it to the output file if specified or to stdout otherwise. The XML syntax of the file closely resembles the internal structure of an SWF file; it is quite verbose and a lot of attention is required to not introduce errors.</td>
</tr>
<tr>
<td>swf2xml &lt;input&gt;</td>
<td>Does the reverse of the previous command, transforming an SWF file to an XML definition. It is very useful because it permits to easily investigate how things are defined in SWF movies.</td>
</tr>
<tr>
<td>xslt &lt;xsl&gt; &lt;input&gt;</td>
<td>The command uses the XSLT file to transform the input XML into another XML document. The resulting XML can be automatically transformed in an SWF file (if the output file has the extension .swf) or maintained as is. The XSLT file can use the swft: extension.</td>
</tr>
<tr>
<td>swf2xml &lt;input&gt;</td>
<td>Uses a simplified XML definition to rapidly define SWF movies. It is the command used in the rest of this section. Note that the simple command only builds SWF movies of version 7. The produced movies can be used in any of the Flash versions targeted by haXe, but they can be used reliably only if they do not implement coded actions. If that is the case, they will work correctly only when used in movies of versions up to 8.</td>
</tr>
</tbody>
</table>
Creating an Image Movie Clip

A simple XML file must contain a root node `movie` that has attributes to define the size and frame rate for the generated movie. Remember that that information only has a meaning when the movie is used alone; when the movie is embedded or loaded into another, the definitions of the container always prevail on the loaded ones. Inside the `movie` element, many other elements can be defined. In the following example an image is loaded using a `clip` element.

```xml
<?xml version="1.0" encoding="iso-8859-1" ?>
<movie width="320" height="240" framerate="12">
  <library>
    <clip id="Picture" import="picture.jpg"/>
    <!-- add more resources here -->
  </library>
  <frame>
    <place id="Picture"/>
  </frame>
</movie>
```

Note the use of the `frame` and `place` element. The first states that the movie contains just one frame and the second places the image visible in the timeline. The `id` attribute has two purposes: referencing the element inside the XML document (`place` creates a reference to the clip using the `id`) and as a class name for the code developers (linkage name). If embedded in haXe, the clip will be available in the code as the class `Picture` that can be enriched with code of its own and that obviously extends the `MovieClip` class.

The `clip` element supports JPGs, PNGs (with or without the alpha channel), SWF movies, and SVG although the implementation of the last is still in an initial phase. A collection of resources can be created using the `library` element. Putting resources in a `library` element is important because it is the only way to have them available in haXe (or ActionScript).

To compile the previous example and the following SWFMill examples, use the following command:

```
> swfmill simple library.xml library.swf
```

Where `library.xml` is the SWFML file to compile and `library.swf` is the output file. The SWMill executable must be on the path of command line/console, and the files described in the XML file must be positioned relatively to the directory where the command is invoked. In the previous example, `picture.jpg` and `library.xml` must be in the same folder where the execution happens.

Using the Image Movie Clip with haXe

How do you use the produced library file in haXe? It is very simple. Create a class with the same linkage name as defined in the XML, add some code to it if necessary, and use your class normally. In Flash versions older than 9, to create an instance of a class of this type, you must use the `attachMovie()` method and cast the returned value to the proper class.

In the following example, the conditional compilation is used to differentiate between Flash 9 and previous versions; the result is the same on all the supported platforms. In the `Picture` class a `startSpin()` method has been created to demonstrate how easy it is to add new methods to the imported clip.
Part II: Server Side, JavaScript, and Flash: Oh My!

#if flash9
class Main
{
    public static function main()
    {
        var pic = new Picture();
        flash.Lib.current.addChild(pic);
        pic.startSpin();
    }
}

class Picture extends flash.display.MovieClip
{
    public function startSpin()
    {
        var self = this;
        onEnterFrame = function() { self._rotation++;};
    }
}#else flash
class Main
{
    public static function main()
    {
        var pic = cast(flash.Lib.current.attachMovie("Picture", "pic", 10), Picture);
        pic.startSpin();
    }
}

class Picture extends flash.MovieClip
{
    public function startSpin()
    {
        var self = this;
        onEnterFrame = function()
        {
            self._rotation++;;
        };
    }
}#end

When compiling the example, remember to add the switch -swf-lib library.swf so that the library file is embedded in the resulting file.

A common mistake is to define a class in a package and to forget to define its full path in the XML file. If the full name of your class is assets.effect.SpinningPicture, the clip attribute must have the following format:

<clip id="assets.effect.SpinningPicture" import="/picture.jpg"/>

Many frames can be defined in a single clip and frames can be individually labeled so that the developer can switch from one to another without necessarily using their indexes or stepping from one frame to the next. In the following example, a hypothetic button is described using a different frame to represent the various visual states that the button can assume. Because the PictureButton is the only class that is
Chapter 13: Replacing the Need for an IDE

also needed in the haXe code, the image clips are set externally to the library and they are invisible outside the XML context.

```xml
<?xml version="1.0" encoding="iso-8859-1" ?>
<movie width="60" height="20" framerate="20">
  <frame>
    <clip id="imgMouseOut" import="picture1.jpg"/>
    <clip id="imgMouseOver" import="picture2.jpg"/>
    <clip id="imgMouseDown" import="picture3.jpg"/>
  </library>
  <clip id="PictureButton">
    <frame name="mouseOut">
      <place id="imgMouseOut" depth="1" />
      <stop />
    </frame>
    <frame name="mouseOver">
      <place id="imgMouseOver" depth="2" />
      <stop />
    </frame>
    <frame name="mouseDown">
      <place id="imgMouseDown" depth="3" />
      <stop />
    </frame>
  </clip>
</movie>
```

The `<stop>` elements are required so that the movie does not advance automatically from one frame to the next. On the haXe side, the code needed to manage the state transition between the different frames can be as simple as the following:

```java
import flash.MovieClip;
class Main {
  public static function main()
  {
    flash.Lib.current.attachMovie("PictureButton", "mc", 1);
  }
}

class PictureButton extends MovieClip {
  public override function onMouseDown() {
    gotoAndStop("mouseDown");
  }
  public override function onMouseUp() {
    // the states for mouseOver and mouseUp are visually equivalent
    gotoAndStop("mouseOver");
  }
  public override function onRollOver() {
  }
```

(continued)
(continued)

gotoAndStop("mouseOver");
}
public override function onRollOut()
{
    gotoAndStop("mouseOut");
}
}

This code only works on Flash up to version 8 because the `simple` command targets the Flash platform version 7. The same effect can also be obtained for Flash 9 but you will have to code a more complex SWFML file and target the latest Flash version.

**Embedding Fonts**

One powerful feature of SWFMill is to embed True Type Fonts (TTF) in movies. You can import an entire set of characters or limit it to a subset using the attribute `glyphs` with the needed characters as value. In this example, a font called Nice Font defined in the file `nicefont.ttf` is included in the produced movie and used directly in SWFMill to create a text field:

```xml
< xml version="1.0" encoding="iso-8859-1" ? >
< movie width="200" height="60" framerate="20" >
    < frame ></ frame >
    < font id="nicefont" name="Nice Font" import="library/nicefont.ttf" />
    < textfield id="firstlabel" font="nicefont" width="200" height="30" size="20"
        text="First Label" useOutlines="1" />
    < place id="firstlabel" name="fl" />
</ movie >
```

The `font` element is not inside a `library` element because otherwise it is necessary to define a class for it in the haXe code. This is not necessary because the font utilization works differently from clips; the fonts are referenced in the haXe code using their internal name, `Nice Font` in this case, and not a class type. In the haXe example, a second text field is defined at run time and the embedded font is assigned to it. The `embedFonts` field must be set to `true` or otherwise the player will try to use a default system font.

```haXe
# if flash9
import flash.text.TextField;
import flash.text.TextFormat;
#else flash
import flash.TextField;
import flash.TextFormat;
# end

class Main
{
    public static function main()
    {
        var lbl : TextField;
        # if flash9
        lbl = new TextField();
        flash.Lib.current.addChild(lbl);
        lbl.y = 30;
        lbl.embedFonts = true;
        lbl.defaultTextFormat = new TextFormat("Nice Font");
    }
}```
User Interfaces

One of the most important and complicated aspects of Flash development is to provide interaction with the user. Interaction can happen in a lot of ways, clicking buttons, inputting texts, making selections, and so on. Several user interface libraries exist to provide standardized sets of configurable controls. The most notable in the Flash world is surely the Adobe Flex library (for more information go to www.adobe.com/products/flex/). Other well-known and quite stable libraries are AsWing (www.aswing.org/) and ActionStep (http://actionstep.org/). Although all of them can be used in conjunction with haXe, there is not a direct way to do that; the libraries must be managed as external resources and there is always extra work to integrate the systems. In the meantime, some new products are appearing in the haXe world and are available at http://lib.haxe.org/. They are still very young and for this reason not as stable and feature rich as the previous ones.

How to Implement an UI

There are literally infinite ways to deal with the user interface problem; every graphic operating system has its own way (or many at once) and every one adopts potentially very different paradigms. In this section you will attempt to build a whole UI library using pure haXe code. The library is built upon the following requirements:

- Platform Independence and Isolation: The haXe nature to be a multi-target environment must be respected. This implies also isolating the mouse and keyboard events from the target system.
- Lightweight: The library must have a reduced footprint in terms of bytes; it must be loaded and displayed fast.
- Basic Set of Controls: Buttons, text inputs, and labels are the very bare minimum required to build interaction.
- Easy Positioning: Some UI API delegate to the developer the responsibility to manage and dispose the controls. Although it is important to have the freedom to create accurate pixel-perfect positioning, in the majority of cases it is more important to be able to dispose of the controls easily and let the system handle the arrangement issues. A set of containers is implemented to ease the visual layout process.
Part II: Server Side, JavaScript, and Flash: Oh My!

- Visually Appealing: Controls must be nice. Products developed with poor graphics are very often labeled as unprofessional works independently from their functionalities. Sad but true.

- Visually Configurable: The aspect of the controls must be easily customizable.

The file structure adopted to build the UI API is schematized in Figure 13-1.
Chapter 13: Replacing the Need for an IDE

The full source code is available at the following address: www.haxe.org/prohaxeandneko or www.wrox.com

To use the source code, download it from one of the locations mentioned and uncompress it in a folder of your choice. To use the included examples, you don’t have to do anything in particular but if you want to use the library in an external project, remember to set a reference to it using the -cp switch in the compiler command or hxml file. The -cp must point a relative or absolute path.

**Events**

The very first problem to deal with is how the events are treated and implemented. haXe does not have a standard API to deal with generic events, but on the haXe site there is a proposal by the very same author of the language, Nicolas Cannasse. The system is dead simple but effective, too: The events are not based on strings and, thus, the compiler can fully check their types for correctness and events can be created with just one line of code. The event proposal can be found at the following address: www.haxe.org/proposals/events.

The main type in the system is the class `Dispatcher <EventType>`; the type argument `EventType` is a definition for an object that is passed as an argument along with the event. In the case of a click event, the `EventType` can be an object of type `MouseEvent` that can contain the coordinates of the cursor on the screen and other mouse-related information. To know more about the event systems, refer to the “Events” section of Chapter 12. Table 13-2 describes the methods for the `Dispatcher` class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addListener(l:Listener &lt;EventType&gt;) :Listener &lt;EventType&gt;</code></td>
<td>A listener is as an object that wraps a function, the handler for the event. The function must be wrapped in an object because function equality is not implemented in all the platforms (Neko does not support it) and the equality operation is required in the <code>removeListener()</code> method to identify the correct reference to remove. The function adds a new <code>Listener</code> and returns it too.</td>
</tr>
<tr>
<td><code>removeListener(l:Listener&lt;EventType&gt;) :Listener&lt;EventType&gt;</code></td>
<td>Removes and returns the <code>Listener</code> argument.</td>
</tr>
<tr>
<td><code>addHandler(f:EventType -&gt; Void) :Listener&lt;EventType&gt;</code></td>
<td>Adds a function handler to the dispatcher and returns it wrapped in a <code>Listener</code> object. The function handler must accept one argument of type <code>EventType</code> and must return nothing.</td>
</tr>
<tr>
<td><code>dispatchEvent(e:EventType) :Bool</code></td>
<td>Invokes the execution of all the registered handlers. The execution chain can be interrupted invoking the <code>stop()</code> method; in that case the function returns <code>false</code> to point out that the execution has been interrupted, otherwise <code>true</code>.</td>
</tr>
<tr>
<td><code>stop() :Void</code></td>
<td>Interrupts the execution of the <code>dispatchEvent()</code> method.</td>
</tr>
</tbody>
</table>
The events that are really important in a UI are mainly those related to the mouse and the keyboard activities. Usually controls implement their event directly with fields such as `onMouseDown` or similar; this practice has the tendency to add a lot of new fields and to bloat the control API. Composition may be the right solution and can aid in maintaining the correct separation of concerns. Because of this, two classes `MouseInteraction` and `KeyInteraction` exist and they are used as a base for platform-specific implementations.

The `MouseInteraction` class defines the events: `click`, `down`, `up`, `move`, `enter`, and `leave`. Their names are very self-explanative because they are associated with a class that represents the mouse actions alone. The `KeyInteraction` just defines `up` and `down`. They both define a static method `replace()` useful to substitute the interaction object with a new one without losing the associated handlers. The mouse and keyboard activities are strictly bound to the underlying platform that in this section is referred to as `context`. Nevertheless, the controls can be declared before being associated to a context; for this reason it is important to be able to define a generic container to support the events and to replace it with the context specifically when this is available. Of course, if the control is not associated with a context, no event is fired even if handlers are already in place.

The `MouseEvent` and the `KeyEvent` are typedef and contain the field definitions described in Tables 13-3 and 13-4.

**Table 13-3**

<table>
<thead>
<tr>
<th>MouseEvent Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target :ui.core.Element</td>
<td>The object that has triggered the mouse event.</td>
</tr>
<tr>
<td>buttonDown :Bool</td>
<td>States if the mouse left button is down or not.</td>
</tr>
<tr>
<td>ctrlKey :Bool</td>
<td>States if the Control key button on the keyboard was pressed at the time that the mouse action has been performed.</td>
</tr>
<tr>
<td>shiftKey :Bool</td>
<td>Same as before but for the Shift key.</td>
</tr>
<tr>
<td>local: { x:Float, y:Float }</td>
<td>Returns the coordinates of the pointer relatives to the target object.</td>
</tr>
<tr>
<td>view: { x:Float, y:Float }</td>
<td>Returns the coordinates of the pointer relatives to the entire view.</td>
</tr>
</tbody>
</table>

**Table 13-4**

<table>
<thead>
<tr>
<th>KeyEvent Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target : ui.core.Element</td>
<td>The object associated to the Dispatcher that has fired the event.</td>
</tr>
<tr>
<td>charCode : Int</td>
<td>The numeric character code for the pressed keyboard button.</td>
</tr>
<tr>
<td>keyCode : Int</td>
<td>The numeric key code for the pressed keyboard button.</td>
</tr>
</tbody>
</table>
Chapter 13: Replacing the Need for an IDE

<table>
<thead>
<tr>
<th>KeyEvent Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char : String</td>
<td>Returns the character selected on the keyboard. Not all the keys can be mapped to a character and they have a null value in that case.</td>
</tr>
<tr>
<td>ctrlKey : Bool</td>
<td>States if the Control key button on the keyboard was pressed at the time that the keyboard has been used.</td>
</tr>
<tr>
<td>shiftKey : Bool</td>
<td>Same as before but for the Shift key.</td>
</tr>
</tbody>
</table>

Views, Controls, Containers, and Elements

A control is a visual object that provides interaction, like a button or an input box. A container usually does not provide any interaction but permits to arrange the contained controls. Controls and containers have some features in common and so they both extend the base abstract Element class. All the elements are hierarchically organized inside a view context. The view is always bound to a target platform, in the current implementation Flash 9 only, and extends the Container class to allow the disposition of its child elements. In practice, the view is the only class connected to the underlying platform that the developer will solely use.

Element

The Element class is the most basic building block for the user interface library and it is included in the ui.core package. The class is abstract and must be extended to be used; the class has no visual representation but it is just a base to build upon. It provides many useful fields for both controls and containers such as the mouse and key fields for intercepting the user gestures. What is really important to understand in the Element class is how its dimensions are calculated and its positioning is managed. The class has no x or y fields; the positioning of the element depends on the selected layouts and on the container that the element belongs to. The layouts are set using the two fields vLayout and hLayout; both variables are of type Layout that is an Enum. The Layout type establishes a behavior for the element relating to how it can stretch or adapt along a dimension. Therefore, the first variable is relative to the vertical behavior and the latter to the horizontal one.

The exact behavior is not just the function of the Layout constructor adopted but also to the container that includes the element; every element must be contained in a container to have a visual representation. There are two base models to interpret the Layout constructors: the bound model and the unbound model (Table 13.5). The first will try to adapt the element layout to a visually defined dimension while the second has no visual constraint and will expand and contract to fit the element dimension.
Table 13-5

<table>
<thead>
<tr>
<th>Layout Constructors</th>
<th>Description for the Bound Model</th>
<th>Description for the Unbound Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Margins(margin: Float, ?marginafter: Null&lt;Float&gt;)</td>
<td>The element will stretch to fit the dimension left subtracting the margins from the total available space. If the second margin is omitted it will be set equal to the first.</td>
<td>The element occupies its fit size and leaves the specified margins before and after (the after value is equal to the before if omitted).</td>
</tr>
<tr>
<td>Center(?content: Null&lt;Float&gt;)</td>
<td>The element has a fixed size (the passed argument or automatically calculated) and it is placed in the center of the available dimension.</td>
<td>The element occupies the specified dimension or, if the argument is omitted, its fit size; no margins are left.</td>
</tr>
<tr>
<td>AnchorStart(margin: Float, ?content: Null&lt;Float&gt;)</td>
<td>The element is anchored at the beginning of the available dimension and occupies the specified size or fit to its content if omitted; if the dimension is not filled completely by the initial margin and the element content a blank space is left after the element.</td>
<td>Same as above but with a specified margin before the element.</td>
</tr>
<tr>
<td>AnchorEnd(margin: Float, ?content: Null&lt;Float&gt;)</td>
<td>Same as above but with the anchor at the end of the available dimension.</td>
<td>Same as above but with the margin at the end.</td>
</tr>
<tr>
<td>Measures(margin: Float, ?content: Null&lt;Float&gt; margin-after:Float)</td>
<td>The margin-before, the content (if omitted the fit size is used) and the margin-after are all specified. The measures of each are calculated proportionally. Measures(20, 50, 30) The above example will render a control that will be rendered at 20% of the beginning margin and occupies the 50% of the available dimension.</td>
<td>The element is rendered with the dimensions specified as arguments: a margin, the content and another margin. If the content size is omitted its fit size is used.</td>
</tr>
</tbody>
</table>

The `Element` class exposes the public fields described in Table 13-6.

Table 13-6

<table>
<thead>
<tr>
<th>Element Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>var vLayout : Layout</td>
<td>The vertical layout.</td>
</tr>
<tr>
<td>var hLayout : Layout</td>
<td>The horizontal layout.</td>
</tr>
<tr>
<td>var minSize : Size&lt;Float&gt;</td>
<td>The minimum size for the element.</td>
</tr>
</tbody>
</table>
Element Fields | Description
---|---
var maxSize : Size<Float> | The maximum size for the element.
var mouse : MouseInteraction | A container for the mouse related events.
var key : KeyInteraction | A container for the keyboard related events.
var context(getContext, null) : IContext | A context object. The context provides the methods to deal with the visual representation of the element and it is platform specific.
var parent (default, null) : Container | A reference to the container for the current element. The parent is null if the control not yet associated to a container. This value is read-only and is responsibility of the container to set it correctly.
function bounds() : Rectangle<Float> | Returns a rectangle definition with the coordinates and dimensions for the current element. The region used to calculate the bounds is established by the container object. If the element is not associated to any container, the function returns null.
function fitSize() : Size<Float> | The fit-size of an element is the ideal dimension for the element. The meaning of ideal can change with the kind of control, but in general it is the minimum required area to fully display the content of the element. The calculation of this size is by the fact responsibility of the rendering engine and thus the context object.
function getView() : IView | Returns the view object that contains the current element or null if the element or its ancestors are not associated to any view.
function getContext() : IContext | If the element is currently associated to a view, returns the context specific object, otherwise null.
function render() : Void | Actually draws the element and its children (if it has any) on the screen.

The `Element` class has references to two very important types: the `IContext` and the `IView`. The first is an interface for classes capable of rendering the elements on the screen; every class that extends `Element` can have a context class that implements `IContext`. The `IView` has just one public string variable `packagename`. The variable is used to build a convention to retrieve the correct context for each element: the class `Button`, when used in a `StageView` (the Flash 9 implementation of the `IView` interface), is rendered by a class with full name `ui.flash9.ButtonContext` (the package name `ui.flash` is defined in the `StageView` class and the `Context` keyword is appended to the class name).

**Controls**

The implemented controls are `Button`, `Label`, and `TextBox`. Out of the rendering context, the three controls are very similar; all use a text caption and they have the same base events for mouse and keyboard interaction. Because of their similarities, they all extend a common class `TextElement` that adds just a `text` variable to the base `Element` class. They differ more in their context and render behaviors.
Part II: Server Side, JavaScript, and Flash: Oh My!

Wiring events to controls is as simple as in the following example:

```javascript
var button = new Button("Button Label");
button.mouse.click.addHandler(function(e) { trace("I've been clicked! "); });
```

Figure 13-2 shows the three controls as rendered by the Flash 9 implementation.

![Figure 13-2](image)

```javascript
import ui.controls.Button;
import ui.controls.Label;
import ui.controls.TextBox;
import ui.containers.GridLayout;
import ui.flash9.StageView;
class Main
{
    public static function main()
    {
        var view = new StageView();
        var grid = new GridLayout(1, 3);
        grid.setElement(new Button("Button"), 0, 0);
        grid.setElement(new TextBox("Text Box"), 0, 1);
        grid.setElement(new Label("Label"), 0, 2);
        view.addChild(grid);
        view.render();
    }
}
```

Containers

The containers are classes that contain and arrange child controls. By default the hLayout and vLayout variables of all the containers are both set automatically to Margins(0). This guarantees that the container occupies all the available area; for the GridLayout this is the only supported configuration in the current implementation.

The children communicate with the container using the `calculateBounds()` method to know what the real area is that they have available for rendering.

PanelLayout

The PanelLayout is the one that gives more flexibility to the developer to position the controls in a precise way. Both the vertical and horizontal directions follow the unbound model. The PanelLayout extends the MultiContainer class the permits to manage an unlimited number of children. Elements are added and removed using the `addChild()`/`removeChild()` methods.
In Figure 13-3, some buttons have been added to a `PanelLayout` and anchored to the view borders. Resizing the view can change the position and the size of the controls but it maintains the distances from the margins.

Figure 13-3
import ui.core.Layout;
import ui.controls.Button;
import ui.flash9.StageView;
class Main {
    public static function main() {
        var view = new StageView();
        var tl = new Button("Top Left");
        tl.vLayout = tl.hLayout = AnchorStart(40);
        view.addChild(tl);
        var tr = new Button("Top Right");
        tr.hLayout = AnchorEnd(40, 100);
        tr.vLayout = AnchorStart(40, 100);
        view.addChild(tr);
        var bl = new Button("Bottom Left");
        bl.hLayout = AnchorStart(40, 100);
        bl.vLayout = AnchorEnd(40, 100);
        view.addChild(bl);
        var br = new Button("Bottom Right");
        br.vLayout = br.hLayout = AnchorEnd(40);
        view.addChild(br);
        var c = new Button("Center");
        c.vLayout = Center();
        c.hLayout = Margins(40);
        view.addChild(c);
        view.render();
    }
}

**StackLayout**

The **StackLayout** arranges the controls in a pile. The pile can be oriented vertically or horizontally as specified in the constructor argument. The class inherits from the **MultiContainer** class. The class has a variable spacing that specifies the distance between two adjacent controls in the pile.

In Figure 13-4 some controls are rendered before in a vertical **StackLayout** and then in a horizontal one.

import ui.core.Layout;
import ui.containers.StackLayout;
import ui.controls.Button;
import ui.flash9.StageView;
class Main {
    public static function main() {
        var view = new StageView();
        // change the following to Horizontal to see the other effect
        var stack = new StackLayout(Vertical);
        view.addChild(stack);
        var first = new Button("First");
        first.hLayout = first.vLayout = Measures(20, 100, 20);
Chapter 13: Replacing the Need for an IDE

```javascript
stack.addChild(first);
var second = new Button("Second");
second.hLayout = second.vLayout = Measures(20, 100, 20);
stack.addChild(second);
var third = new Button("Third");
third.hLayout = third.vLayout = Measures(20, 100, 20);
stack.addChild(third);
view.render();
```
import ui.core.Layout;
import ui.controls.Button;
import ui.flash9.StageView;
import ui.containers.GridLayout;
class Main
{
    public static function main()
    {
var view = new StageView();
var t = 4;
var grid = new GridLayout(t, t);
grid.hLayout = grid.vLayout = Margins(10);
grid.spacing = 5.0;
view.addChild(grid);
for(r in 0...t)
    for(c in 0...t)
    {
        var b = new Button(Std.string(c+r*t+1));
b.hLayout = b.vLayout = Margins(0);
grid.setElement(b, r, c);
    }
view.render();

FieldLayout

The FieldLayout accepts just one Element instance. The purpose of this control is to associate a Label, which is automatically generated, to another element. Piling some fields vertically can be useful to have the controls in them perfectly aligned.

The Flash 9 Implementation

The Flash 9 implementation of the user interface is not really complicated. It simply maps the requirements from the UI controls and containers to object specifics for the environment. The ButtonContext has three visual states for the mouseover, mouseout andmousedown combinations. The TextBoxContext has just two states, one when the control has the focus and the caret is visible and one when it is not selected. The LabelContext has just one state. It is possible to change their aspect acting on the static fields with prefix style and change some of the available parameters.

The StageView uses the whole stage area to dispose child controls. It extends the PanelLayout and, thus, can accept many children directly. When the stage is resized, the children are re-rendered automatically; this can happen if the movie is opened in the standalone player or embedded in a web page with the movie dimensions set with percentage values.

A Complete Example: the Tree Editor

To complete the section with a full example, a user interface is built upon the tree generator class from the previous chapter. The controls are used to change the parameters of the generator and two buttons are responsible to invoke the generation procedure and to clear the drawing area. The controls are contained in a StackLayout aligned to the right of the screen. Each input control is contained in a double FieldLayout to define a proper label for the input and its unit of measurement.
import ui.core.Layout;
import ui.containers.StackLayout;
import ui.containers.FieldLayout;
import ui.controls.Button;
import ui.controls.Label;
import ui.controls.TextBox;
import ui.flash9.StageView;
import Tree;

class TreeEditor
{
    public static function main()
    {
        var application = new TreeEditor();
        application.run();
    }
    var blength : TextBox;
    var bangle   : TextBox;
    var bwidth   : TextBox;
    var branches : TextBox;
    var bcolor   : TextBox;
    public function new() { }
    public function run()
    {
        var view = new StageView();
        var panel = new StackLayout(StackDirection.Vertical);
        panel.spacing = 5;
        panel.hLayout = AnchorEnd(10, 210);
        view.addChild(panel);
        blength =  new TextBox("100");
        bangle =   new TextBox("45");
        bwidth =   new TextBox("8");
        branches = new TextBox("3");
        bcolor =   new TextBox("0x000000");
        var fldw = 80.0;
        vartxtw = 50;
        panel.addChild(new Label("TREE PROPERTIES"));
        panel.addChild(new FieldLayout(fldw, "Branch Length:",
            new FieldLayout(txtw, "px", blength, Right)));
        panel.addChild(new FieldLayout(fldw, "Branch Angle:",
            new FieldLayout(txtw, "°", bangle, Right)));
        panel.addChild(new FieldLayout(fldw, "Branch Width:",
            new FieldLayout(txtw, "px", bwidth, Right)));
        panel.addChild(new FieldLayout(fldw, "Branches:",
            new FieldLayout(txtw, "qt", branches, Right)));
        panel.addChild(new FieldLayout(fldw, "Color:",
            new FieldLayout(txtw, "hex", bcolor, Right)));
        var gbtn = new Button("Generate");
gbtn.hLayout = Margins(0, 0);
gbtn.vLayout = Measures(0, 60, 0);
Chapter 13: Replacing the Need for an IDE

gbtn.mouse.click.addHandler(generateTree);
panel.addChild(gbtn);
var cbtn = new Button("Clear");
cbtn.hLayout = Margins(0, 0);
cbtn.mouse.click.addHandler(clear);
panel.addChild(cbtn);
view.render();
}
private function generateTree(e)
{
  var s = flash.Lib.current.stage;
  var tree = new Tree(flash.Lib.current.graphics);
  tree.angle = getAngle(bangle, 25);
  tree.branches = getInt(branches, 3, 1, 10);
  tree.color = getInt(bcolor, 0x000000, 0x000000, 0xffffffff);
  //x, y, len, angle, width
  tree.generate(s.stageWidth/2-105, s.stageHeight,
  getInt(blength, 80, 10, 200), Angle.NORTH, getInt(bwidth, 8, 2, 16));
}
private function getInt(el : TextBox, alt : Int, min : Int, max : Int) : Int
{
  var v = Std.parseInt(el.text);
  if(v != null && v >= min && v <= max)
  {
    return v;
  } else {
    el.text = Std.string(alt);
    return alt;
  }
}
private function getAngle(el : TextBox, alt : Int) : Float
{
  var v = getInt(el, alt, -180, 180);
  return v/180*Math.PI;
}
private function clear(e)
{
  flash.Lib.current.graphics.clear();
}

The result is shown in Figure 13-6.
Conclusions

The proposed implementation covers only the Flash 9 platform. All the classes that are Flash-specific are limited to a unique package. Writing the connection layer between another platform, say Flash 8 or JavaScript, and the UI API is just a matter of creating a new specific package on the same line as the existing one.

About being lightweight, the Tree Editor example is around 12.6KB, a quite good result after all considering that in Flex introducing just a single button will produce a file of around 115KB. The confrontation with Flex is unfair because of the very different level of features implemented and API richness, but sometimes being light is as important as being rich. The generated files are small because they do not rely on embedded or loaded resources; all the graphic elements are generated dynamically using the drawing API.

Using the style static field it is possible to change a lot of the aspects of the controls in their different states. The current implementation does not permit radical changes such as adding a background image.
but tweaking the `render()` method a little bit is not hard at all and it is possible to eventually write more than one context for each control.

The system can really be improved. Just follow a list of suggestions in case you think you can extend the basic system to fit your needs:

- Adding new controls — checkboxes, combo boxes, select controls, tree lists, sliders, and scrolling bars are good candidates.
- Adding new containers like a `FlowLayout` (objects are disposed from left to right and from top to bottom when one line is not enough to fit them all) or windows and message boxes, and improving on the existing ones; span functionality for the `GridLayout` for example.
- Adding features to elements like enabled/disabled and visible/hidden.
- Adding new contexts for Flash older versions and JavaScript.
- Adding specialized views; having a view that maps on a 3D object could be very nice.
- Rendering optimization for the current Flash 9 implementation.
- Extending the event model to support more interactions.

**Summary**

In this chapter, a new tool was added to your toolset enhancing the possibilities of integrating more contents in your products. The user interface problem was also discussed and a proposal for a possible implementation was introduced. The following arguments were introduced and explained in detail:

- How to use `SWFMill` to embed contents in a SWF file
- How haXe can interact with the embedded resources
- What the requirements are for a good user interface library
- How to implement a platform-independent UI library.

In the next chapter, it is time to target a different platform and to see how haXe can be proficiently used to enhance HTML/JavaScript-based documents.
More Interactive Content with JavaScript

When properly used, the JavaScript language can be a very powerful tool to enhance the user interaction with web pages. This chapter shows how to use haXe to produce useful scripts and to implement AJAX. This chapter reviews the following topics:

- What JavaScript can do
- Which tools can help the developer to work better
- What the haXe JavaScript API is
- How to validate a form on the client side
- How to use AJAX with haXe

Dynamic Content in JavaScript?

Chapter 9 discussed how to generate dynamic contents for web pages using Neko; the server receives a request and produces a suitable response that is sent back to the client. This is perfect when the requested document provides no further interaction or the interaction is limited to navigating to other pages. Nowadays sites are more and more interactive pieces of software; they require much more than hyperlink navigation. The Asynchronous JavaScript and XML (AJAX) and the Web 2.0 paradigms are just concepts that highlight the necessity to provide richer interactivity to the user and to develop better interfaces.

So how can you build richer and more interactive pages? The short answer is JavaScript. A web page is composed (or at least should be composed) of three distinct layers:

- **Content layer**: This is provided by the HTML document that defines the text contents, the references to external resources, and the structure.
Part II: Server Side, JavaScript, and Flash: Oh My!

- **Presentations layer**: This is provided by one or more associated style sheets commonly in the Cascading Style Sheet (CSS) format.

- **Behavior layer**: This is provided by one or more script blocks usually written in JavaScript language that is the only language currently available in all the modern web browser software.

The presentation layer and the behavior layer should always be made optional. Even if disabled or removed, the page content should be displayed in a meaningful way. The presentation and behavior layer must be provided to enhance, sometimes considerably, the navigation experience and are not contents on their own.

The behavior layer can be used to achieve an unlimited number of results acting on the other layers or separately from them. Table 14-1 shows the more common problems that the behavior layer is called to resolve without the need for reloading the whole page.

### Table 14-1

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lazy Loading</td>
<td>A portion of the document is loaded only if the user requires it. A classical example of it is having a list (or a table) of records where the user selects one of them and a detailed and richer block of information is displayed near there.</td>
</tr>
<tr>
<td>Data Update</td>
<td>An element on the page requires being refreshed several times during the page lifecycle. It can be a block on a web page that displays stock-quotes or live information on arrivals and departures of trains/airplanes and so on. The data update can occur on schedule or by user request.</td>
</tr>
<tr>
<td>Data Completion</td>
<td>The UI provides hints to the user to speed-up the data input. A search box can provide the auto-completion of recognized keywords.</td>
</tr>
<tr>
<td>Data Manipulation</td>
<td>The content is manipulated dynamically to provide a better experience to the user. The user can sort, filter, or paginate a set of records in a list or table to obtain a better view on the displayed data.</td>
</tr>
<tr>
<td>Validation</td>
<td>The script filters and validates the contents provided by the user before their submission to the server.</td>
</tr>
<tr>
<td>Computation</td>
<td>The web page manipulates the data provided by the user and gives an immediate response. A unit measure converter, a currency converter, or a simple calculator are all examples of computation problems that can be resolved on the client side.</td>
</tr>
<tr>
<td>Visual Effects</td>
<td>The script affects the presentation of the page: elements can change in style (colors, fonts, borders, and so on), can be hidden or displayed, can be moved or resized.</td>
</tr>
</tbody>
</table>
Chapter 14: More Interactive Content with JavaScript

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced Controls</td>
<td>The behavior layer provides functionalities and controls that are not provided by the base implementations of the browser vendors. Date/Time selectors, combo boxes, and sliders are examples of controls that can be created using JavaScript.</td>
</tr>
<tr>
<td>User Preferences</td>
<td>User preferences can be stored and retrieved by the behavior layer. Using the browser cookies the scripts can have access to small pieces of data stored on the client side. Those may be used to remember useful settings when changing page in the same domain or getting back to the site in a subsequent navigation session.</td>
</tr>
</tbody>
</table>

### The Web Developer Tools

Developing for the client side can be tricky. To maximize the opportunities to investigate and debug your applications, it is suggested that you use proper tools. Although many IDEs exist, both commercial and open source, that effectively target the JavaScript development, there are some tools that are easy to obtain because of open source that cannot miss in your development toolbox.

- **Firefox**: Is a cross-platform (Windows, Mac OSX, Linux and almost any operative system with a graphical front end) and its implementation is consistent in every environment. It is very flexible because it allows it to be enhanced using plug-ins known as extensions. The Firefox developers are very careful to implement and to be on par with the latest standard released and particularly the standards coming from the World Wide Web Consortium (W3C). Firefox is released under the Mozilla Public License (MPL) and can be downloaded at [www.mozilla.com/firefox/](http://www.mozilla.com/firefox/)

- **Firebug**: Is the second tool of choice and it is an extension for Firefox. “Firebug integrates with Firefox to put a wealth of web development tools at your fingertips while you browse. You can edit, debug, and monitor CSS, HTML, and JavaScript live in any web page.”—Description from the Firebug site. It has been realized by Joe Hewitt and is available at [www.getfirebug.com/](http://www.getfirebug.com/)

- **Web Developer**: Is another Firefox extension. The Web Developer extension adds a menu and a toolbar to the browser with various web developer tools. Amongst the other functionalities it permits to disable temporarily the JavaScript execution and the application of style sheets, to edit the CSS definitions at run time, and to see the alterations applied on the fly, to resize the browser window to the desired dimensions. The extension has been created by Chris Pederick and can be downloaded at [http://chrispederick.com/work/web-developer/](http://chrispederick.com/work/web-developer/)

### Trace to Firebug

The haXe library includes a class to properly redirect the trace commands to the Firebug console. This can be used also when developing Flash movies in a web page context and the ExternalInterface feature is available. To redirect the trace command you can use the following snippet of code:

```java
class Main
{
    public static function main()
        (continued)
```
Part II: Server Side, JavaScript, and Flash: Oh My!

(continued)

```haxe
if(haxe.Firebug.detect())
    haxe.Firebug.redirectTraces();
    trace("Hello!");
}
```

The `haxe.Firebug.detect()` returns `true` if the Firebug extension is available and the `haxe.Firebug.redirectTraces()` simply set the `trace` function to use the `Firebug.trace()` function. The preceding code produces the result shown in Figure 14-1.

![Firebug Console](image)

Figure 14-1

When developing, remember to activate the `-debug` switch in your compilation command. In JavaScript each function will be marked with its position and other information that will be very useful when developing and an error occurs. The `-debug` adds extra bytes and slightly affects the performances but it can be removed before deploying your scripts.

**JavaScript as haXe Target**

When the haXe compiler targets the JavaScript environment, a single `.js` file is produced containing a bootstrap section and the code introduced by the developer. This can be a little confusing for the JavaScript developer that is often accustomed to importing several files each with its own purpose.

Remember though that haXe produces just one file because the module/functionality selection is made at compile time and only the needed definitions are exported in the output, keeping the resulting file lean and easy to deploy (one file ready to go).

The JavaScript produced by the haXe compiler is meant to be human readable and it is not optimized for small footprint size. If you are not satisfied with the size of the output, you can find a lot of useful tools that can reduce it considerably and that can even obfuscate it to discourage reverse engineering. If you are in need of such a tool, search for `javascript compressor` or `javascript minification` on your Internet search engine of choice. Another way to improve the web performances is to set your web server to send the files compressed with the commonly available gzip format.
Chapter 14: More Interactive Content with JavaScript

The code examples in this chapter have been tested with the following browsers:

- Mozilla Firefox 2.0.0.6
- Opera 9.23
- Safari 3.0.3
- Microsoft Internet Explorer versions 5.5, 6, and 7

The JavaScript Library

The haXe standard library contains a js package dedicated to the JavaScript environment. The package includes type definitions to ease the communication between haXe and the hosting environment, usually the web browser.

When the haXe compiler targets JavaScript the produced file always contains a base set of instructions. That minimal piece of code is very light (about 13KB) and provides a solid base to build upon that includes the following definitions: Std, IntIter, js.Boot, js.Lib, and some initialization calls. Other definitions are added as far as they are referenced in the user code.

The Std class is discussed in Chapter 3 and in other chapters, and the IntIter is described in Chapter 4; both are common classes available for every supported haXe target. The other two classes are from the js package; the first is to initialize the JavaScript environment and should not be used directly from the developer. The second provides some useful features frequently used. Table 14-2 describes the methods and variables of the js.Lib class.

Table 14-2

<table>
<thead>
<tr>
<th>js.Lib Static Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>function alert(v:Dynamic) : Void</td>
<td>Calls the alert() JavaScript function.</td>
</tr>
<tr>
<td>function eval(code:String) : Dynamic</td>
<td>Evaluates and processes the code passed as an argument. Note that the code must be JavaScript code and not haXe code.</td>
</tr>
<tr>
<td>function setErrorHandler (f:String-&gt; Array&lt;String&gt;-&gt;Bool) : Void</td>
<td>Allows redirecting the errors to a custom function. The custom function must have as arguments a text message and an array containing the messages from the execution stack and must return a Boolean value. If the function returns true, the error is ignored; otherwise the execution is interrupted.</td>
</tr>
<tr>
<td>var window : Window</td>
<td>Is a reference to the Window object in a DOM tree.</td>
</tr>
<tr>
<td>var isIE : Bool</td>
<td>Returns true if the current browser is Internet Explorer.</td>
</tr>
<tr>
<td>var isOpera : Bool</td>
<td>Returns true if the current browser is Opera.</td>
</tr>
</tbody>
</table>
Part II: Server Side, JavaScript, and Flash: Oh My!

Although the `Lib` class provides the `isIE` and `isOpera` variables, it is often a better practice to test the browser for the needed feature and not rely on the detected name of the agent.

### Redirecting Errors to a Custom Function

In the following example, the execution errors are redirected to a custom function that simply traces the error messages and does not break the code execution. Avoiding the errors in this way is not really a good practice; changing the return value to `false`, the execution is stopped and the browser reports an error to the user.

```haxe
class Main
{
    public static function main()
    {
        js.Lib.setErrorHandler(function(msg, arr)
        {
            trace(msg + ", " + Std.string(arr));
            return true;
        });
        throw "Error!";
    }
}
```

### HTML DOM

The Document Object Model (DOM) is a language-independent representation of an HTML/XML document. Elements in the DOM are organized in a hierarchy of nodes. The DOM adopted by haXe is the one defined by the W3C and tries to avoid mapping definitions available only on certain browsers. The idea is to be as standard as possible. The DOM mapping does not add any functionality or tries to resolve any compatibility issue; it is straight mapping, admittedly with some dark areas of features not implemented, of the W3C definition. The advantage for the developer is that he has all the flexibility to build what he needs from scratch and the output file is not encumbered with potentially unused features. The biggest disadvantage is that he or she will have to solve browser-specific issues when encountered.

The package includes a file `Dom.hx` that maps the definitions of the HTML Elements to haXe types. Figure 14-2 shows the definitions included in the file. Note that those definitions are not external classes but just plain `typedef` and, thus, they cannot be created using the standard instantiation statement but by using the proper methods provided by the DOM environment. The `MetaDom<T>` definition, an abstract definition of a DOM node, and the `Dom` that is a shortcut for the type `MetaDom<Dom>` have been omitted from the schema.

Table 14-3 provides a short description of each `typedef` declaration. A detailed description of properties and methods can be found on the haXe site in the API documentation area, and the W3C site provides an area dedicated to the DOM technology full of useful information. Descriptions have been omitted for definitions that map directly to an HTML element with the same name.
Chapter 14: More Interactive Content with JavaScript

Figure 14-2
### Table 14-3

<table>
<thead>
<tr>
<th>Dom typedef</th>
<th>Extends</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anchor</td>
<td>HtmlDom</td>
<td>A definition that corresponds to an <code>&lt;a&gt;</code> element in HTML. The anchor can bring information about a hyperlink relation either as a target or as a trigger.</td>
</tr>
<tr>
<td>Body</td>
<td>HtmlDom</td>
<td>-</td>
</tr>
<tr>
<td>Button</td>
<td>FormElement</td>
<td>A button control. The corresponding elements are <code>&lt;input type=&quot;checkbox&quot; /&gt;</code> or <code>&lt;button&gt; &lt;/button&gt;</code>.</td>
</tr>
<tr>
<td>Checkbox</td>
<td>FormElement</td>
<td>A checkbox input control. The corresponding element is <code>&lt;input type=&quot;checkbox&quot; /&gt;</code>.</td>
</tr>
<tr>
<td>Document</td>
<td>HtmlDom</td>
<td>Represents the HTML document as a whole. It is a very important definition because all the possible interactions with the DOM tree must start with it.</td>
</tr>
<tr>
<td>Event</td>
<td>-</td>
<td>A definition that brings information about a triggered event like the pointer coordinates and the status of the keyboard keys. It is automatically created and passed to every event handler (fields that are prefixed with <code>on</code>).</td>
</tr>
<tr>
<td>FileUpload</td>
<td>FormElement</td>
<td>Represents an input control that permits the upload of a single file to a web server. The corresponding element is <code>&lt;input type=&quot;file&quot; /&gt;</code>.</td>
</tr>
<tr>
<td>Form</td>
<td>HtmlDom</td>
<td>-</td>
</tr>
<tr>
<td>FormElement</td>
<td>HtmlDom</td>
<td>A base definition for all of the input controls used in a form.</td>
</tr>
<tr>
<td>Frame</td>
<td>HtmlDom</td>
<td>-</td>
</tr>
<tr>
<td>Frameset</td>
<td>HtmlDom</td>
<td>-</td>
</tr>
<tr>
<td>Hidden</td>
<td>FormElement</td>
<td>A hidden input control; the control passes its value on form submission but it does not have a visual representation. The corresponding element is <code>&lt;input type=&quot;hidden&quot; /&gt;</code>.</td>
</tr>
<tr>
<td>History</td>
<td>-</td>
<td>The definition for an object that contains information about the user navigation history. The object also allows navigating backward and forward between the already accessed pages.</td>
</tr>
</tbody>
</table>
# Chapter 14: More Interactive Content with JavaScript

<table>
<thead>
<tr>
<th>Dom typedef</th>
<th>Extends</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HtmlCollection &lt;T&gt;</td>
<td>-</td>
<td>A collection of T nodes. It can be accessed as an array using the square bracket syntax.</td>
</tr>
<tr>
<td>HtmlDom</td>
<td>MetaDom &lt;HtmlDom&gt;</td>
<td>Defines fields specific to an HTML element such as id and className that corresponds to the class attribute in an HTML document.</td>
</tr>
<tr>
<td>IFrame</td>
<td>HtmlDom</td>
<td>-</td>
</tr>
<tr>
<td>Image</td>
<td>HtmlDom</td>
<td>An &lt;img /&gt; element.</td>
</tr>
<tr>
<td>Location</td>
<td>-</td>
<td>An object that represents the current URI location. It permits client-side redirects, reloading and accessing information such as the hostname and the current path.</td>
</tr>
<tr>
<td>MetaDom&lt;T&gt;</td>
<td>-</td>
<td>Allows navigating and management of the relations between nodes with fields like parentNode, childNodes, and appendChild().</td>
</tr>
<tr>
<td>Navigator</td>
<td>-</td>
<td>Provides information over the browser used to navigate the page.</td>
</tr>
<tr>
<td>Option</td>
<td>FormElement</td>
<td>-</td>
</tr>
<tr>
<td>Password</td>
<td>FormElement</td>
<td>A password input control. The corresponding element is &lt;input type=&quot;password&quot;/&gt;.</td>
</tr>
<tr>
<td>Radio</td>
<td>FormElement</td>
<td>A radio button control. The corresponding element is &lt;input type=&quot;radio&quot;/&gt;.</td>
</tr>
<tr>
<td>Reset</td>
<td>FormElement</td>
<td>A button that can restore the default status of the whole form. The corresponding element is &lt;input type=&quot;reset&quot;/&gt;.</td>
</tr>
<tr>
<td>Screen</td>
<td>-</td>
<td>Contains information about the user screen as the resolution and the color depth.</td>
</tr>
<tr>
<td>Select</td>
<td>FormElement</td>
<td>-</td>
</tr>
<tr>
<td>Style</td>
<td>-</td>
<td>Contains no methods; the definition properties describe and alter the visual aspect of an element. Some variables only affect certain types of elements.</td>
</tr>
<tr>
<td>StyleSheet</td>
<td>-</td>
<td>An element that contains or that references a CSS definition. It is the type for elements like &lt;link rel=&quot;stylesheet&quot; type=&quot;text/css&quot;/&gt; or &lt;style type=&quot;text/css&quot;&gt;&lt;/style&gt;.</td>
</tr>
</tbody>
</table>

*Table continued on following page*
Part II: Server Side, JavaScript, and Flash: Oh My!

<table>
<thead>
<tr>
<th>Dom typedef</th>
<th>Extends</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Submit</td>
<td>FormElement</td>
<td>A submit button control; it is different from a standard button because it is automatically associated to the form submission action. The corresponding element is <code>&lt;input type=&quot;submit&quot; /&gt;</code>.</td>
</tr>
<tr>
<td>Text</td>
<td>FormElement</td>
<td>A text input control. The corresponding element is <code>&lt;input type=&quot;text&quot; /&gt;</code>.</td>
</tr>
<tr>
<td>Textarea</td>
<td>FormElement</td>
<td>-</td>
</tr>
<tr>
<td>Window</td>
<td>-</td>
<td>Contains fields related to the host window that is showing the web page.</td>
</tr>
</tbody>
</table>

As already mentioned, the current implementation is far from being exhaustive but is enough to start. The JavaScript environment is by now the one less explored in the haXe community but certainly has great potential.

**Low-Level Access to JavaScript**

When a developer builds core scripts that must access otherwise unavailable constructs in JavaScript or scripts that must be optimized for execution speed, he or she can always rely on the untyped blocks and the `__js__()` function. The function, that must be invoked in an untyped block, accepts a string of pure JavaScript code; unlike the `js.Lib.eval()` function that is evaluated at run time, the argument of the `__js__()` function is embedded as is” in the output file. Remember that an untyped block is never checked for type correctness and the argument in `__js__()` is not checked at all, thus, it is very easy to introduce subtle errors in them; use with care!

**Manipulating the Page with JavaScript**

To change a document it is necessary to obtain a reference to the node you intend to modify. This can be done by navigating the children of the body field of the `js.Lib.document` object or more likely by using the method `getElementById()` of the Document typedef. This method returns a reference to an element using a string identifier to locate it; if no element matches the past identifier, a null reference is returned. Once you obtain such a reference you can change the element contents working on the `innerHTML` field or create new child nodes. The `HtmlDom`, the type associated to every element returned by using the `getElementById()` method, also contains fields to query and to change the style of the element, its attributes, and its events.

**The OnLoad Problem**

The DOM hierarchy can only be fully manipulated once it has been completely loaded. The page is downloaded and displayed progressively and the DOM tree is assembled as far as complete elements have been parsed. The only way to be sure to traverse the tree without receiving errors, is to wait until the full page is complete. This implies two possible solutions: including the `script` block at the end of the page as the very last element or acting on the page load event. The first solution is practical but tends to be unclear for JavaScript developers who prefer to put their `script` tags only in the head
Chapter 14: More Interactive Content with JavaScript

A Possible Approach for a Cross-Browser Event System

Before implementing an onload solution, what it is needed is to overcome the event handling differences. The most basic approach to events is to redefine the element methods prefixed with on. It is possible to perform an action on the click event of a button just by writing something like the following:

```javascript
import js.Dom;
class Main
{
    public static function main()
    {
        var btn : Button = cast js.Lib.document.getElementById("sbutton");
        btn.onclick = function(e) { js.Lib.alert("Button Clicked!"); };
    }
}
```

The element retrieved by `getElementById()` must be cast to the Button type because the function returns an object of HtmlDom type that does not contain the definition for the onclick event (actually the event is from FormElement but because btn is a Button it makes more sense this way). The cast is of an unsafe type because it is not possible to use a proper cast on a typedef.

The container HTML page can be as simple as the following example. However, note how the script element is inside the body element at the end of the page.

```html
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <button id="sbutton">Display Message</button>
    <script type="text/javascript" src="main.js"></script>
  </body>
</html>
```

The solution in the preceding example suffers a very big limitation: The same event can be associated to one and only one function at a time. This may not be a problem most of the time but you must be sure that your script has exclusive access to those methods or unexpected cancellations may occur. A better way to implement events is to use the `addEventListener()` DOM method. This method is not supported on every modern browser (see Internet Explorer before version 7), so it must be wrapped in a compatibility layer. One possible solution is the following code snippet:

```javascript
import js.Dom;
class DomListener
{
    public static function add(el : Dynamic, t : String, f : Event -> Void)
    {
        if(Reflect.hasField(js.Lib.window, "addEventListener"))
            el.addEventListener(t, f, false);
        else
        {
            if(Reflect.hasField(js.Lib.window, "attachEvent"))
                el.attachEvent(t, "function(e) { f(e); }");
            else
                return false;
        }
    }
}

var btn : Button = cast js.Lib.document.getElementById("sbutton");
btn.add('click', js.Lib.alert("Button Clicked!"));
```

(continued)
Part II: Server Side, JavaScript, and Flash: Oh My!

(continued)

\[
\begin{align*}
\text{add} &= \text{function}(\text{el}, \ t, \ f) \\
\quad &= \text{el}.\text{addEventListener}(t, \ f, \ false); \\
\text{else} \ {} \\
\text{add} &= \text{function}(\text{el}, \ t, \ f) \\
\quad &= \text{el}.\text{attachEvent}(\text{on} + t, \ f); \\
\end{align*}
\]

\[
\begin{align*}
\text{add}(\text{el}, \ t, \ f); \\
\end{align*}
\]

\[
\begin{align*}
\text{public static function remove}(\text{el} : \text{Dynamic}, \ t : \text{String}, \ f : \text{Event} \to \text{Void}) \\
\quad &= \text{function} \\
\quad &= \text{if}(\text{Reflect}.\text{hasField}(\text{js.Lib}.\text{window}, \ \text{"removeEventListener"})) \\
\quad &= \text{function}(\text{el}, \ t, \ f) \\
\quad &= \text{el}.\text{removeEventListener}(t, \ f, \ false); \\
\text{else} \ {} \\
\quad &= \text{function}(\text{el}, \ t, \ f) \\
\quad &= \text{el}.\text{detachEvent}(\text{on} + t, \ f); \\
\text{remove}(\text{el}, \ t, \ f); \\
\end{align*}
\]

The `add()` method simply adds a new listener to the element passed as the first argument; the argument is the `Dynamic` type because not only objects of type `HtmlDom` support events, but the `Window` object supports them, too. The event is determined by a string identifier passed as the second argument; to know what events exist for every HTML element, refer to the events described in the HTML Appendix and strip the `on` prefix. The third argument is the function that acts as a handler of the event; the function accepts an `Event` object as an argument and returns nothing. The code shows a good technique to lessen the differences between browsers; first of all no browser detection has been performed; instead a feature detection technique has been adopted: If the method `addEventListener()` on the `window` object does not exist, then the `addEventListener()` DOM method is not implemented at all and another solution must be encountered. The second good practice is to perform the test just once and not every time the function is accessed. To do that, on the first execution, the current method is replaced by one that is adapted to the current browser and then immediately executed. Finally, note the necessity to wrap the code in an `untyped` block to access low-level JavaScript features.

Now that you have a standard way to add and remove events, it is possible to write code that is executed after the page loading completion to satisfy the JavaScript purists. This is very easy to accomplish now, just add an event listener to the load event of the `window` object and put the initialization code in there.
Chapter 14: More Interactive Content with JavaScript

```javascript
import DomListener;
import js.Dom;

class Main {
    public static function main()
    {
        DomListener.add(js.Lib.window, "load", function(e) {
            var btn : Button = cast js.Lib.document.getElementById("sbutton");
            DomListener.add(btn, "click", function(e) {
                js.Lib.alert("Button Clicked!");
            });
        });
    }
}
```

The script can now be positioned correctly in the `head` element of the page.

```html
<html>
<head>
    <title>Example</title>
    <script type="text/javascript" src="main.js"></script>
</head>
<body>
    <button id="sbutton">Display Message</button>
</body>
</html>
```

To be fair, the page load event is not the best place to start the DOM manipulation. This is because the page is considered loaded not when the whole HTML has been downloaded but after the page has loaded and all the references to images, styles, and embedded objects have been loaded, too. Because this can create a considerable delay before the user interface is ready to capture the user gestures, this can have a strong impact on the user interaction. To avoid the problem, some browsers as Firefox and Opera define a proper event called `DOMContentLoaded` but it is not implemented in other browsers. Anyway, it is possible to simulate the event in various ways; you can have a look at the most popular JavaScript libraries listed next to see how the problem has been faced and resolved.

**Advanced JavaScript Libraries**

To improve your JavaScript skills, consider trying one or more of the following open source JavaScript libraries if you have not already done it; they are really mines of ideas and eventually they can be wrapped in haXe to be used as they are.

- **dojo**: “Dojo is an Open Source DHTML toolkit written in JavaScript. It builds on several contributed code bases (nWidgets, Burstlib, f(m)), which is why we refer to it sometimes as a “unified” toolkit. Dojo aims to solve some long-standing historical problems with DHTML, which prevented mass adoption of dynamic web application development.” Description from the official website. The library is licensed under the Academic Free License v2.1 license and is supported by the nonprofit organization Dojo Foundation. Website: [http://dojotoolkit.org/](http://dojotoolkit.org/)
Part II: Server Side, JavaScript, and Flash: Oh My!

- Ext: This is a client-side, JavaScript framework for building web applications. In early 2006, Jack Slocum began working on a set of extension utilities for the Yahoo! User Interface (YUI) Library that rapidly evolved to an independent solution under the name of Ext. The library is feature rich and provides beautiful widgets out of the box; the code is very well organized and extending it further is very easy. Ext is dual-licensed under the LGPL (Lesser General Public License) and a commercial license. Website: http://extjs.com/

- jQuery: This is a library with a long history of successes and is supported by a very strong community. The main author of the library is John Resig and is released under the dual-license MIT and GPL. Description from the official homepage: “jQuery is a fast, concise JavaScript Library that simplifies how you traverse HTML documents, handle events, perform animations, and add AJAX interactions to your web pages. jQuery is designed to change the way that you write JavaScript.” Website: http://jquery.com/

- Mochikit: “MochiKit is a highly documented and well tested, suite of JavaScript libraries that will help you get things done, fast. We took all the good ideas we could find from our Python, Objective-C, etc. experience and adapted it to the crazy world of JavaScript.” Description from the official website. It is dual-licensed under the MIT license and the Academic Free License v2.1. Website: http://mochikit.com/

- mootools: The library is well documented and released under the MIT style license. Its main author is Valerio Proietti. Description from the official homepage: “MooTools is a compact, modular, Object-Oriented JavaScript framework designed for the intermediate to advanced JavaScript developer. It allows you to write powerful, flexible, and cross-browser code with its elegant, well-documented, and coherent API. MooTools code respects strict standards and doesn’t throw any warnings. It’s well commented and has meaningful variable names: a joy to browse and a snap to understand.” Website: www.mootools.net/

- Prototype: The library was written initially for the AJAX support in Ruby on Rails by Sam Stephenson and has grown fast in an independent solution. It is released under the MIT license. Description from the official homepage: “Prototype is a JavaScript Framework that aims to ease development of dynamic web applications. Featuring a unique, easy-to-use toolkit for class-driven development and the nicest AJAX library around, Prototype is quickly becoming the code-base of choice for web application developers everywhere.” Website: www.prototypejs.org/

- script.aculo.us: This is an add-on for Prototype that provides features like animation, drag and drop, AJAX controls, DOM utilities, and unit testing. One of the reasons of its success is the integration with the Ruby on Rails framework. The library is licensed under the MIT license and has been created by Thomas Fuchs. Website: www.script.aculo.us/

- YUI: “The Yahoo! User Interface (YUI) Library is a set of utilities and controls, written in JavaScript, for building richly interactive web applications using techniques such as DOM scripting, DHTML, and AJAX. The YUI Library also includes several core CSS resources. All components in the YUI Library have been released as open source under a BSD license and are free for all uses.” Description from the official website: http://developer.yahoo.com/yui/

Traversing HTML with JavaScript

A DOM tree can be navigated by reference, as already explained using the Document.getElementById() method, by direct relationship using the MetaDom fields or yet by filtering a subset of elements using the getElementsByTagName() of the HtmlDom definition. To illustrate how the DOM can be traversed and changed dynamically, it is proposed a TableSorter class.
Table Sorting

A standard table element in an HTML page can be enhanced adding sorting capabilities. The sorting action is triggered clicking on the heading of each column; the default sorting direction is ascending but on the second and further clicks, the direction is reversed. The columns are ordered by default using a string comparison, the behavior can be changed using the `setParser()` function that accept the column index as the first argument and a function that converts a string value to `Dynamic` as the second argument. When the table values are parsed (just once on the first order action), they are stored in an array, the private `data` field. Each element in the array contains a reference to the row in the DOM tree, the `tr` element, and an array of values that are the contents of each cell, the `td` elements; those values are transformed by the parser function if it exists and are used for sorting purposes.

The HTML page does not indicate that it will be enhanced by the `TableSorter` class and will render normally if, for example, the user has disabled the JavaScript execution in his browser.

```html
<html>
<head>
  <title>Countries</title>
</head>
<body>
<table id="countries">
  <thead>
    <tr><th>Country</th><th>Population</th><th>National Holiday</th></tr>
  </thead>
  <tbody>
    <tr><td>China</td><td>1,321,851,888</td><td>Anniversary of the Founding of the People's Republic of China, 10-1 (1949)</td></tr>
    <tr><td>India</td><td>1,129,866,154</td><td>Republic Day, 1-26 (1950)</td></tr>
    <tr><td>European Union</td><td>490,426</td><td>Europe Day, 5-9 (1950)</td></tr>
    <tr><td>United States</td><td>301,139,947</td><td>Independence Day, 7-4 (1776)</td></tr>
    <tr><td>Indonesia</td><td>234,693,997</td><td>Independence Day, 8-17 (1945)</td></tr>
    <tr><td>Brazil</td><td>190,010,647</td><td>Independence Day, 9-7 (1822)</td></tr>
    <tr><td>Pakistan</td><td>164,741,924</td><td>Republic Day, 3-23 (1956)</td></tr>
    <tr><td>Bangladesh</td><td>150,448,339</td><td>Independence Day, 3-26 (1971)</td></tr>
    <tr><td>Russia</td><td>141,377,752</td><td>Russia Day, 6-12 (1990)</td></tr>
    <tr><td>Nigeria</td><td>135,031,164</td><td>Independence Day (National Day), 10-1 (1960)</td></tr>
  </tbody>
</table>
<script type="text/javascript" src="main.js"></script>
</body>
</html>

Part II: Server Side, JavaScript, and Flash: Oh My!

The interesting thing in the table is that the second column is numeric and sorting must be made in this disguise; the third column is trickier, it contains a text description, a date portion (month-day), and the anniversary year in parentheses. The order on that column must follow the month-day sequence because that is probably what the user may expect by the heading “National Holiday.”

To address the second column, the `setParser()` function is invoked passing a function that strips off the thousand separators and parses the resulting string to an `Int` value. For the third column, the string is matched against a regular expression that extracts just the month and day values; the month is multiplied by one hundred and added to the day to obtain an integer that can be easily compared.

To enhance a table, a reference to it must be passed as argument to the method `apply()`.

```javascript
import js.Dom;
class Main {
    public static function main() {
        var sorter = new TableSorter();
        sorter.setParser(1, function(v) {
            return Std.parseInt(StringTools.replace(v, ',', ''));
        });
        var re = ~/\s(\d+)-(\d+)\s\(/;
        sorter.setParser(2, function(v) {
            if(re.match(v))
                return Std.parseInt(re.matched(1)) * 100 + Std.parseInt(re.matched(1));
            else
                return 0;
        });
        sorter.apply(js.Lib.document.getElementById("countries"));
    }
}
```

The `apply()` method first locates the table head, `thead`, and the table body, `tbody`, then adds an event handler of the `onclick` method of each column heading; the event when triggered, executes `sortTable()` function. This function receives the index of the column that is used to order the data. The `getData()` function collects the data from the table on the first attempt of sorting. The function uses a first invocation of `getElementsByTagName()` to retrieve all the rows, `tr`, in the table, and then for each row a new invocation is used to retrieve the individual cells, `td`. To grab the content cell, the `innerHTML` property is used. This property is also very useful to change the content of an element; the new value must be a string containing a valid HTML fragment. Note, however, that certain elements do not work as expected with `innerHTML` in Internet Explorer (the `tbody` of a table cannot be set in this way for example).

```javascript
import js.Dom;
type Row = { tr : HtmlDom, values : Array<Dynamic> };
type DataGrid = Array<Row>;
class TableSorter {
    private var order : { index : Int, asc : Bool };
    private var tbody : HtmlDom;
    private var parsers : Array<String -> Dynamic>;
    private var data : DataGrid;
```
Chapter 14: More Interactive Content with JavaScript

public function new()
{
    parsers = new Array();
}

public function apply(table : HtmlDom)
{
    if(table == null)
        throw "InvalidTable";
    var thead = table.getElementsByTagName("thead") [0];
    tbody = table.getElementsByTagName("tbody") [0];
    if(thead == null || tbody == null)
        throw "InvalidTableFormat";
    order = { index : -1, asc: true };
    data = null;
    var ths = thead.getElementsByTagName("th");
    var self = this;
    for (i in 0...ths.length)
        untyped ths[i].onclick = function(e) { self.sortTable(i); };
}

public function setParser(index : Int, f : String -> Dynamic)
{
    parsers[index] = f;
}

private function getData() : DataGrid
{
    if(data == null)
    {
        data = new DataGrid();
        var trs = tbody.getElementsByTagName("tr");
        for (i in 0...trs.length)
            {
                var tds = trs[i].getElementsByTagName("td");
                data[i] = { tr: trs[i], values: new Array<Dynamic>() };
                for (j in 0...tds.length)
                    data[i].values[j] = if (parsers[j] != null)
                        parsers[j](tds[j].innerHTML)
                    else
                        tds[j].innerHTML;
            }
    }
    return data;
}

private function sortTable(index : Int)
{
    changeOrder(index);
    sortRows();
}

private function changeOrder(index : Int)
{
    if(order.index == index)
        order.asc = !order.asc;
    else
    {
        order.index = index;
        (continued)
Part II: Server Side, JavaScript, and Flash: Oh My!

(continued)

```
order.asc = true;
}
}
private function sortRows()
{
    var o = order;
    var data = getData();
data.sort(function(a : Row, b : Row)
{
    return {
        if     (a.values[o.index] < b.values[o.index]) -1
        else if(a.values[o.index] > b.values[o.index]) 1
        else 0
    ) * (o.asc ? 1 : -1);
});
for(row in data)
tbody.appendChild(row.tr);
}
```

The example can be compiled using the following command:

```
> haxe -main Main -js main.js
```

**Data Validation in HTML Forms**

The HTML syntax has not evolved a lot from its first definition in the area of form submissions and input controls. The available tools such as text inputs, checkboxes, and buttons are essentially the same since then. The working draft definitions for the next generation standards such as (X)HTML5, Web Forms 2.0, XHTML 2, XForms 1.0, and complementary technologies are very promising, but at the moment very few browsers support them in a complete way and those technologies are not usable out of the box. In the meantime, while waiting for the browsers to embrace the new standards, problems must be resolved with the available tools.

One of the biggest issues in HTML forms is having a good feedback about the correctness of the user actions. There are controls that limit a lot of the interaction with the users and this is of great help to them because it helps to avoid mistakes. A `select` element, for example, only permits to choose from a limited set of values; introducing errors is quite unfeasible. What happens if your system must receive an input from the user that must be an arbitrary number? Having a `select` with every possible number is very unlikely. The way provided by the HTML syntax is to use an `input` element of type `text`. However, what guarantees that the input value is effectively a number? Well, nothing. That is why the server should always adopt some validation mechanism and provide a suitable answer to the clients that have tried to submit an invalid set of data. JavaScript can enhance the user experience a lot in the area of form interactions. Providing client-side validation is a sure way to give to the user a faster and better feedback.

Remember that the client validation can never replace the server validation. Client validation is a mean to give a better feedback to the user and not a way to protect your system from abuses; there are many ways to send data to a server skipping the client-side verification, disabling the JavaScript support is probably the most simple. So, if you are short on time and you have to decide between server validation and client validation, wonder no more!
Chapter 14: More Interactive Content with JavaScript

The client/server-side validation is one of the few areas where the DRY principle can be violated at least in part; haXe with its multiplatform target capabilities, permits reuse of good portions of the code on both the client and the server, the only differences being platform specific.

With the topic of the separation of concerns in mind, which is the layer in which the client validation is to be defined? The validation process is for sure about contents; not the contents that are usually maintained and served by a server, but contents that are introduced by a user with the intent to process them is some way. It is not by luck that the `<input type="text" />` supports an optional argument to define the maximum allowed characters. This is a primitive way of validating contents on the client side. New standards define a lot more of this kind of attribute for the HTML elements. It seems an obvious consequence to put the validation rules in the HTML context. A rule is the definition of a constraint that must be applied to a control. If the HTML syntax had an `input` of type `email`, it would be obvious to understand the rule that is applied on the control and which values can satisfy it.

One way to introduce the rules in an HTML document could be using the `class` attribute. Classes define groups of elements semantically related so it seems to be a good place to put the validation rules; on the other side class is quite limited about the values that can be expressed within it. You can say that a control can have class `email`, but how do you express that the control is a number with a value between 5 and 10? Defining a class for each possible combination could be a pain considering that in many cases those rule parameters can be optional.

A possible solution is to extend the HTML. One of the nice things that all the HTML browsers have in common is that if they do not recognize an element or an attribute, they simply ignore it. Of course, adding a nonstandard attribute will break the HTML validation and for the purists this will be simply inadmissible. If you cannot live with this solution, you can always move the validation rule binding to the behavior layer, but this will probably violate the separation of concern paradigm because you are moving contents to the wrong place.

To limit the syntax invalidation to the bare minimum, just one new attribute is added: `validation`. The `validation` attribute defines the rules to apply to the current control and their optional parameters. More than one rule can be applied to each control separated with one whitespace character. The example shows an input control for a value that is mandatory and that only accepts Internet e-mail addresses as values.

```
<input type="text" name="email" validation="required email" />
```

The attribute expresses its purposes in a very clean and easy way. An input control for a password that must be between 6 and 20 characters long could be defined in this way.

```
<input type="password" name="password" validation="required range(min:6,max:20)" />
```

The named parameters help a lot to identify the purpose of each argument.

To keep the implementation simple, the validation rule is working reliably only on `input` elements of type `text` and `password`; some work may be needed to make it work on other elements.

What follows is a full example of a form with some rules applied. Figure 14-3 shows the example rendered by the Mozilla Firefox browser (the CSS style sheet content is linked in the example but is not provided here for brevity).
<html>
<head>
<title>Form Validation</title>
<link rel="stylesheet" type="text/css" href="main.css" />
</head>
<body>
<form action="#" method="post">
<fieldset>
<legend>Personal Information</legend>
<div>
<label for="name">Full Name</label>
<input type="text" name="name" id="name" validation="required" />
</div>
<div>
<label for="age">Age</label>
<input type="text" name="age" id="age" validation="int(min:13,max:150)" />
</div>
</fieldset>
<fieldset>
<legend>Login Information</legend>
<div>
<label for="email">Email</label>
<input type="text" name="email" validation="required email" />
</div>
<div>
<label for="username">User Name</label>
<input type="text" name="username" validation="required loweralphanum range(min:3,max:12)" />
</div>
<div>
<label for="password">Password</label>
<input type="password" name="password" validation="required range(min:6,max:20)" />
</div>
<div>
<label for="confirmpassword">Confirm Password</label>
<input type="password" name="confirmpassword" validation="compare(field:password)" />
</div>
</fieldset>
<fieldset class="submission">
<input type="reset" name="reset" value="Reset" />
<input type="submit" name="submit" value="Submit" />
</fieldset>
<script type="text/javascript" src="main.js"></script>
</form>
</body>
</html>
Chapter 14: More Interactive Content with JavaScript

The form at the moment has nothing special. If the Submit button is pressed, it is submitted to the server as usual whether it is valid or not. It is needed to provide a way to interpret and to apply the rules of the validation attributes. The last line before the closing of the body element includes a JavaScript file that is where the validation logic is executed. Note that the content layer and the behavior layer do not make any assumption on the used rules; they both work only on the convention that a certain rule can exist with a certain name and parameters. If in the form a new field is added with a validation attribute, there is no need to change the behavior layer unless a rule that is not already defined is introduced.

The haXe code to compile the main.js file must contain the definition of the rules and a mechanism to apply the rules to the controls. The FormValidator class is responsible for managing one form at the time enhancing the controls it contains with the validation behaviors. It accepts a Form element as argument for the constructor but probably the most obvious way to use the class is to invoke its static applyToDocument() method that will create a new instance of FormValidator for each form that exists in the page. The rules are registered using the static method registerRule(); this way a registered rule works for every instance of FormValidator. The registration requires a name (the one used in the validation attribute), a class that is used to manage the rule, and an optional object that defines the default parameters. A default parameter may be overridden by a parameter with the same name defined in the HTML document.

```haxe
import FormValidator;
import ValidationRule;

class Main
{
    public static function main()
    {
        FormValidator.registerRule("required", RequiredRule);
    }
}
```

Figure 14-3

The form at the moment has nothing special. If the Submit button is pressed, it is submitted to the server as usual whether it is valid or not. It is needed to provide a way to interpret and to apply the rules of the validation attributes. The last line before the closing of the body element includes a JavaScript file that is where the validation logic is executed. Note that the content layer and the behavior layer do not make any assumption on the used rules; they both work only on the convention that a certain rule can exist with a certain name and parameters. If in the form a new field is added with a validation attribute, there is no need to change the behavior layer unless a rule that is not already defined is introduced.

The haXe code to compile the main.js file must contain the definition of the rules and a mechanism to apply the rules to the controls. The FormValidator class is responsible for managing one form at the time enhancing the controls it contains with the validation behaviors. It accepts a Form element as argument for the constructor but probably the most obvious way to use the class is to invoke its static applyToDocument() method that will create a new instance of FormValidator for each form that exists in the page. The rules are registered using the static method registerRule(); this way a registered rule works for every instance of FormValidator. The registration requires a name (the one used in the validation attribute), a class that is used to manage the rule, and an optional object that defines the default parameters. A default parameter may be overridden by a parameter with the same name defined in the HTML document.

```haxe
import FormValidator;
import ValidationRule;

class Main
{
    public static function main()
    {
        FormValidator.registerRule("required", RequiredRule);
    }
}
```
Part II: Server Side, JavaScript, and Flash: Oh My!

(continued)

```haxe
class ValidationRule
{
    public var error(default, null) : String;
    public var params(default, null) : Hash<Dynamic>;
    public var context(default, null) : Dynamic;
    private function new(context : Dynamic)
    {
        this.context = context;
        params = new Hash();
    }
    public function validate(value : String)
    {
        return throw "AbstractMethod";
    }
    private function isValid()
```

In the example are defined the rules used in the above HTML page. More can be defined reusing the available ValidationRule subclasses or by defining new classes, that satisfies specific validation requirements.

ValidationRule is an abstract class to validate a single value. The validate() function performs the check and returns true if the value has been validated successfully. If the validation fails, the reasons are stored in the error field and used by the context to inform the user. The argument passed to validate() is a string because this is the only data type possible when using values from a form. Note, however, how the implementation of ValidationRule is independent from the FormValidator class and it is possible to use it and its subclasses in other programming contexts like Neko or Flash. The only one exception is the CompareRule class that is platform specific; it must communicate with the form to check that the value to validate is equal to the one of another control. This communication happens using the context field whose value is set at the moment of the instantiation of the rule. The context is dynamic because it is platform specific and can change if the same system is implemented in a different environment.

The following code illustrates the implementation of the ValidationRule and of some concrete subclasses. They are all in the same ValidationRule.hx file but in a real-world application, they probably can be located in a proper package and each class in its own file.

```haxe
    }
}
```

```haxe
FormValidator.registerRule("int", NumericRule,
    { useint : true });
FormValidator.registerRule("email", PatternRule,
    { pattern : "^([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})$",
        options : 'i',
        message : "Invalid email format" });
FormValidator.registerRule("loweralnum", PatternRule,
    { pattern : "^[a-z0-9]+$",
        message : "Use only lower-case, alpha-numeric characters" });
FormValidator.registerRule("range", RangeRule);
FormValidator.registerRule("compare", CompareRule);
FormValidator.applyToDocument();
}
```
Chapter 14: More Interactive Content with JavaScript

```javascript
private function isInvalid(message : String)
{
    error = message;
    return false;
}
}

class NumericRule extends ValidationRule
{
    private function useint()
    {
        return if(params.exists("useint"))
        Std.bool(params.get("useint"));
        false;
    }

    public override function validate(value : String)
    {
        if(value == '')
            return isValid();
        if(useint())
            {
                if(!~/^\d+(\d{1,3}(,\d{3})*))?$/i.match(value))
                    return isInvalid("Value must be an integer");
            } else {
                if(!~/^\d+(\d{1,3}(,\d{3})*)(\.\d+)?$/i.match(value))
                    return isInvalid("Value must be a number");
            }
        var v = Std.parseFloat(value);
        if(params.exists("min") && v < Std.parseFloat(params.get("min")))
            return isInvalid("Value must be at least " + params.get("min");
        if(params.exists("max") && v > Std.parseFloat(params.get("max")))
            return isInvalid("Value must be no more than " + params.get("max");
        return isValid();
    }
}

class RangeRule extends ValidationRule
{
    public override function validate(value : String)
    {
        if(value == '')
            return isValid();
        if(params.exists("min") && value.length < Std.parseInt(params.get("min")))
            return isInvalid("Value must be at least " + params.get("min") + " characters long");
        if(params.exists("max") && value.length > Std.parseInt(params.get("max")))
            return isInvalid("Value must be no more than " + params.get("max") + " characters long");
        return isValid();
    }
}

(continued)
```
Part II: Server Side, JavaScript, and Flash: Oh My!

(continued)

class CompareRule extends ValidationRule
{
    public override function validate(value : String)
    {
        var validator = cast(context, FormValidator);
        var field = params.get("field");
        if(validator == null)
            trace("Null validator instance");
        else if(field == '')
            trace("No comparison field specified");
        else if(validator.getControlValue(field) != value)
            return isInvalid("Fields do not match");
        return isValid();
    }
}
class RequiredRule extends ValidationRule
{
    public override function validate(value : String)
    {
        if(value != '')
            return isValid();
        else
            return isInvalid("Required Field");
    }
}
class PatternRule extends ValidationRule
{
    public override function validate(value : String)
    {
        if(value == '')
            return isValid();
        var opt = if(params.exists("options")) params.get("options") else "";
        var pattern = if(params.exists("pattern")) params.get("pattern") else "";
        var re = new EReg(pattern, opt);
        if(re.match(value))
            return isValid();
        else
            return isInvalid(if(params.exists("message"))
                params.get("message")
            else
                "Field doesn't match the pattern");
    }
}

The FormValidator does different things. First, it wires the two events of the form onsubmit and onreset. The first triggers a global validation action preventing the submission if at least an error has occurred; the second, whose purpose is to restore the form to its initial state, adds a cleanup function that removes any error message currently visible. Then the constructor traverses every input control in the form and looks for the presence of a validation attribute. If the attribute exists, it is parsed and interpreted; a new rule is created and stored along with a reference to the control. Each control is also enhanced with an onblur handler that is executed each time the user moves the focus from the current control to another. When this happens, the validation check is executed but only on the control that has
lost the focus and an error is reported eventually. In this way, the user does not have to wait until the final submission to be informed if he has made some mistakes. The class is also responsible for managing (displaying and removing) error messages.

import js.Dom;
class FormValidator
{
  private static var rules : Hash< { rclass:Class<ValidationRule>,
    params : Dynamic } > = new Hash();
  private var fields : Hash< { el : FormElement, rules : Array<ValidationRule> } >;
  public function new(form : Form)
  {
    form.onsubmit = submit;
    form.onreset = reset;
    var inputs = form.getElementsByTagName("input");
    fields = new Hash();
    var self = this;
    for(i in 0...inputs.length)
    {
      var crules = new Array();
      var validation = inputs[i].getAttribute("validation");
      if(validation != null)
      {
        for(r in parseRules(validation))
        {
          if(!rules.exists(r.name))
          {
            trace("Rule not registered: "+r.name);
            continue;
          }
          var v = rules.get(r.name);
          var rule : ValidationRule = Type.createInstance(v.rclass, [this]);
          for(key in Reflect.fields(v.params))
            rule.params.set(key, Reflect.field(v.params, key));
          for(key in r.params.keys())
            rule.params.set(key, r.params.get(key));
          crules.push(rule);
        }
        untyped inputs[i].onblur = function(e)
        {
          self.validateControl(inputs[i].getAttribute("name"));
        }
      }
      fields.set(inputs[i].getAttribute("name"),
        { el : cast inputs[i], rules : crules });
    }
  }
  public static function applyToDocument()
  {
    for(i in 0...js.Lib.document.forms.length)
      new FormValidator(js.Lib.document.forms[i]);
  }
}
public static function registerRule(name : String, 
   rule : Class<ValidationRule>, ?params : Dynamic)
{
   if(params == null)
   params = Reflect.empty();
   rules.set(name, { rclass : rule, params : params });
}
private function validateAll()
{
   var valid = true;
   for(name in fields.keys())
      if(!validateControl(name))
         valid = false;
   return valid;
}
private function resetAll()
{
   for(name in fields.keys())
      cleanErrorMessage(name);
}
private function validateControl(name)
{
   var valid = true;
   var errors = new Array();
   for(rule in fields.get(name).rules)
   {
      if(!rule.validate(getControlValue(name)))
      {
         valid = false;
         errors.push(rule.error);
      }
   }
   if(!valid)
      createErrorMessage(name, errors);
   else
      cleanErrorMessage(name);
   return valid;
}
public function getControlValue(name : String)
{
   return fields.get(name).el.value;
}
private function getErrorMessageId(name)
{
   return name + "_error_message";
}
private function createErrorMessage(name, messages)
{
   var id = getErrorMessageId(name);
   var el = js.Lib.document.getElementById(id);
Chapter 14: More Interactive Content with JavaScript

if(el == null)
{
  el = js.Lib.document.createElement("div");
  el.id = id;
  el.className = "validation-error-message";
  fields.get(name).el.parentNode.appendChild(el);
}

el.innerHTML = "<ul>
    <li>" + messages.join("<li>") + "</li>
</ul>";
}
private function cleanErrorMessage(name)
{
  var el = js.Lib.document.getElementById(getErrorMessageId(name));
  if(el != null)
    el.parentNode.removeChild(el);
}
private function parseRules(v : String)
{
  var rules = v.split('');
  var a = new Array();
  for(rule in rules)
  {
    var p = rule.split('');
    var name = StringTools.trim(p[0]);
    var params = new Hash();
    if(p.length > 1)
    {
      var sparams = StringTools.trim(p[1]);
      sparams = sparams.substr(0, sparams.length-1);
      var spairs = sparams.split(' ,');
      for(spair in spairs)
      {
        var pair = spair.split(' :');
        params.set(StringTools.trim(pair[0]), StringTools.trim(pair[1]));
      }
    }
    a.push({ name : name, params : params });
  }
  return a;
}
private function submit(e : Event)
{
  return validateAll();
}
private function reset(e : Event)
{
  resetAll();
}

A user that tries to submit an invalid form is blocked by the validation check. Figure 14-4 illustrates such a case.
Part II: Server Side, JavaScript, and Flash: Oh My!

The weakest part of the current implementation is probably the `parseRules()` function that is responsible for transforming a string containing the validation rules into some variables. When you use a more sophisticated parsing logic, maybe some regular expressions, it is possible to implement a solution that permits the inclusion of separator characters (like spaces, commas, and parentheses) inside the string parameters without breaking the parsing rule. The current implementation has been chosen to favor the code readability and shortness.

**What Is AJAX?**

AJAX is a technique that implements communication between an HTML page and a host server without the need for refreshing the whole page. The technique is advantageous because the size of the exchanged data is smaller than a whole page and the user interface is more responsive. The name *Asynchronous JavaScript and XML* is derived from the technical implementation of this technique using the `XMLHttpRequest` (XHR) object. In reality, the communication is not constrained to the XML format and can actually use any text format such as JavaScript Object Notation (JSON) or other custom text formats.
Chapter 14: More Interactive Content with JavaScript

Removing the Myths Around AJAX Complexity

Developers that have never used AJAX may fear its use as painful and indeed it can be on some platforms; that is not the case in haXe, which has been built with communication in mind since day one. The JavaScript library provides a class XMLHttpRequest that is a low-level abstraction over the communication object implemented in the different browsers. Although it is possible to work using this class, it is not the easiest and most effective way. A better choice is to use the haxe.Http class that is a cross-platform implementation available for every supported platform (Table 14.4).

Table 14-4

<table>
<thead>
<tr>
<th>haxe.Http Public Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>function setHeader( header:String, value: String ) : Void</td>
<td>Sets a value in the communication header.</td>
</tr>
<tr>
<td>function setParameter( param:String, value: String ) : Void</td>
<td>Sets a parameter that is submitted with the request.</td>
</tr>
<tr>
<td>function setPostData( data:String ) : Void</td>
<td>Neko and JavaScript only. Sets the whole content of a POST request.</td>
</tr>
<tr>
<td>function request( post:Boolean ) : Void</td>
<td>Performs the configured request. The method does not return anything. To monitor the communication results use the onData, onError, and onStatus methods. The post argument states if the HTTP method to use is POST or GET.</td>
</tr>
<tr>
<td>f9dynamic function onData( data:String ) : Void</td>
<td>The event that is fired once the connection has completed the data transfer. The data argument contains the result of the communication.</td>
</tr>
<tr>
<td>f9dynamic function onError( msg:String ) : Void</td>
<td>The event that is fired when a communication error occurs. The msg argument contains the reason of the failure.</td>
</tr>
<tr>
<td>f9dynamic function onStatus( status:Int ) : Void</td>
<td>Fires each time there is an alteration in the connection state. The status code is passed as an argument.</td>
</tr>
</tbody>
</table>

One of the easiest applications of the AJAX technique is to update the content of an element in an HTML document. To obtain the effect, two components must be considered: the server side and the client side. On the server side goes code that produces a fragment of HTML. On the client side a placeholder is needed in the page for the requested data that is usually a simple div element with a proper id and some code to make the request and manage the response. Neko is used for the generation of the web services on the server; although highly recommended, this is not the only possible way to proceed. Experienced programmers can use other server side languages to provide the same functionalities.
**Updating a Panel Content Using AJAX**

In the following example, a page contains an empty `div` element with `id quotation`. Once the page is loaded the client makes a second request to the server to obtain a random quotation and puts the response inside the placeholder element. There can be many reasons why the quotation is not served directly in the page: It can come from a different service, the panel can be updated several times during the page lifetime, or it is possible to introduce interaction with the user that can request a new quotation on the click of a button.

First the quotation service is implemented using Neko. The class `QuotationService` is very basic: When the script is executed, Neko opens a text file that contains all of the quotations, reads them all, and picks one up randomly. After that, it formats the quotation in an HTML fragment and sends it as the response to the request. To see how to compile and work with Neko on the server side, see Chapter 9.

```javascript
import neko.io.File;
typedef Quotation = { author : String, quotation : String }
class QuotationService
{
    public static function main()
    {
        var q = pickRandom(loadQuotations());
        neko.Lib.print(toHtmlFragment(q));
    }
    private static function toHtmlFragment(q)
    {
        var b = new StringBuf();
        b.add(' < q > ');
        b.add(q.quotation);
        b.add(' < /q >  ( < em > ');
        b.add(q.author);
        b.add(' < /em > )');
        return b.toString();
    }
    private static function loadQuotations()
    {
        var c = StringTools.trim(File.getContent("quotations.txt"));
        var q = new Array();
        for(line in c.split("\n"))
        {
            var p = line.split("|");
            q.push({ author:StringTools.trim(p[0]), quotation:StringTools.trim(p[1]) });
        }
        return q;
    }
    private static function pickRandom(qs : Array<Quotation>)
    {
        var index = Std.int(Math.random() * qs.length);
        return qs[index];
    }
}
```
Chapter 14: More Interactive Content with JavaScript

The file contains a quotation per line. Each line is composed by the author name, a pipe character (|), and the quotation text. You can test the script by compiling it to a quotation.n file and invoking it, pointing your browser to your development web server or executing the file on the command line.

The client page can be a complex home or as simple as the one in the following code. The importance is the presence of the placeholder element with id="quotation" and of course of the JavaScript file.

```html
<html>
<head>
  <title>Quotation Example</title>
</head>
<body>
  <div id="quotation"></div>
  <script type="text/javascript" src="main.js"></script>
</body>
</html>
```

Now it is time for the client-side script. It is amazing how compact and simple it is.

```haxe
import js.Dom;
import haxe.Http;
class Main {
  public static function main() {
    var cnx = new Http("/quotation.n");
    cnx.onData = function(data) {
      var div = js.Lib.document.getElementById("quotation");
      div.innerHTML = data;
    }
    cnx.onError = function(msg) { js.Lib.alert("Error: " + msg); }
    cnx.request(false);
  }
}
```

Building an Auto-Complete Control

The previous example was quite basic. Of course there are more advanced uses for the AJAX technique and one that is usually requested because it is really useful in the implementation of an auto-complete control. An auto-complete control is a text input control that eases the data input suggesting valid values. The suggestion uses the first characters input to filter a set of possible values. The more characters are typed by the user, the narrower the filter result will be. While the list of possible values can be very long, it is not a good idea to send it with the requesting page; instead an AJAX request is performed every time a certain number of characters has been typed, thus reducing the communication to only a subset of all the available possibilities.

The following is an example that illustrates an implementation of an auto-completion control. The control in question is used to select a color by its name. The color names are stored on the server and a service outputs all of the colors that have a name that begins with the passed argument string. The response is an XML document that contains the name of the colors that satisfy the query and their hex
values. The control shows the queried colors in a list below the input control and a panel that visualizes
the selected color. The effect is illustrated in Figure 14-5.

Once more the HTML page is very simple and not aware that it will be enhanced by a script.

```html
<html>
<head>
  <title>Quotation Example</title>
  <link rel="stylesheet" type="text/css" href="main.css" />
</head>
<body>
  Type the Color Name: <input type="text" id="colorinput" name="colorinput" />
</body>
</html>
```

There is a reference to a style sheet that is used to obtain the effect illustrated in the figure. The CSS is out
of the scope of this chapter but the illustrated aspect can be reproduced easily using the proper class
and id selectors.

The client and the server scripts share a common piece of code contained in the class ColorsUtil that is
a utility class defined to help the developer to stay DRY. The class has three methods to transform a list
of colors to and from an XML document and to filter a collection of colors using the first letters of their
names.

```typescript
typedef Color = {
  name : String,
  hex : String
}
class ColorsUtil
{
  public static function fromXml(data : String)
  {
    var xml = Xml.parse(data);
    var colors = new Array();
    for(color in xml.firstChild())
      colors.push({name : color.get("name"), hex : color.get("hex") });
    return colors;
  }
  public static function toXml(list : Iterable<Color>)
  {
    // implementation
  }
}
```

418
Chapter 14: More Interactive Content with JavaScript

```javascript
var xml = Xml.createDocument();
var colors = Xml.createElement("colors");
xml.appendChild(colors);
for(item in list)
{
    var color = Xml.createElement("color");
    color.set("name", item.name);
    color.set("hex", item.hex);
    colors.appendChild(color);
}
return xml.toString();
}
public static function filterByName(start : String, list : Iterable<Color>)
{
    start = start.toLowerCase();
    return Lambda.filter(list, function(c)
    {
        return StringTools.startsWith(c.name.toLowerCase(), start);
    });
}
```

The server script is also very simple and resembles the QuotationService from the previous example. The ColorsService reads a list of color names and hex definitions from a text file and filters the result using the optional parameter start.

```java
import neko.io.File;
class ColorsService
{
    public static function main()
    {
        var start = neko.Web.getParams().get("start");
        if(start == null)
            start = '>';
        var colors = loadColors();
        neko.Lib.print(ColorsUtil.toXml(ColorsUtil.filterByName(start, colors)));}
    private static function loadColors()
    {
        var c = File.getContent("colors.txt");
        var q = new Array();
        for(line in c.split("\n"))
        {
            var p = line.split("|");
            q.push({ name : StringTools.trim(p[0]), hex : StringTools.trim(p[1]) });
        }
        return q;
    }
}
```

The file with the color definitions contains one description per line with the name of the color and its hex value separated by a pipe character (|). To find a list of valid colors, just make a search on your preferred search engine using the keywords css color names.
Part II: Server Side, JavaScript, and Flash: Oh My!

The client main script just instantiates an object of type `RemoteColorSelector`. The class takes as an argument the id value of the placeholder element on the page.

```javascript
class Main{
    public static function main()
    {
        new RemoteColorSelector("colorinput");
    }
}
```

The class constructor creates the container for the list of filtered colors, known in the code as `tags`, and the color sample area. The input control is enhanced with a key-up event. Each time the control has the focus and the keyboard is used, the `update()` method checks if a new list must be loaded, refreshes the list of visible tags and eventually the selected color. The color tags can also be used to select a color using the mouse.

Some untyped blocks have been introduced to bypass browser incompatibilities, as is the case of the control text selection in the `selectColor()` function.

```javascript
import js.Dom;
import ColorsUtil;
class RemoteColorSelector {
    private static var min = 1;
    private static var serviceurl = "colors.n";
    private var last : String;
    private var input : Text;
    private var eltags : HtmlDom;
    private var elsampler : HtmlDom;
    private var colors : Array<Color>;
    public function new(id : String)
    {
        input = cast js.Lib.document.getElementById(id);
        createTagsContainer();
        createColorSampler();
        last = '';
        input.onkeyup = update;
    }
    private function update(e)
    {
        var ev = if(e == null) untyped window.event else e;
        var text = input.value.toLowerCase();
        if(text.length == 0 || text == last)
            return;
        // backspace and delete keys must not refresh the color selection
        refreshTags(ev.keyCode == 8 || ev.keyCode == 46);
        if(text.length < min)
            return;
        if(last == '' || !StringTools.startsWith(text, last.substr(0, min))){
            var cnx = new haxe.Http(serviceurl);
            cnx.onData = loadList;
        }
    }
}
```
Chapter 14: More Interactive Content with JavaScript

```
cnx.setParameter('start', text.substr(0, min));
cnx.request(false);
}
last = text;
}
private function loadList(data : String)
{
    colors = ColorsUtil.fromXml(data);
    refreshTags(false);
}
private function refreshTags(skipSelect : Bool)
{
    if(colors == null || input.value == '')
        return;
    var filtered = ColorsUtil.filterByName(input.value, colors);
    eltags.innerHTML = '';
    var self = this;
    for(color in filtered)
    {
        var tag = cast js.Lib.document.createElement("a");
        untyped tag.onclick = function(e) {
            self.selectColor(color);
            return false;
        };
        tag.innerHTML = ' < span style="background-color:'+
                        color.hex+'"> &nbsp;&nbsp;&nbsp; &nbsp; '+color.name;
        eltags.appendChild(tag);
        if(color != filtered.last())
            eltags.appendChild(untyped js.Lib.document.createTextNode(', '));
    }
    if(!skipSelect && filtered.length > 0) {
        selectColor(filtered.first());
    }
}
public function selectColor(c : Color)
{
    elsampler.style.backgroundColor = c.hex;
    untyped
    {
        if(input.setSelectionRange)
        {
            var pos = input.selectionStart;
            input.value = c.name;
            input.setSelectionRange(pos, input.value.length);
        } else if(input.createTextRange){
            var pos = input.value.length-document.selection.createRange().text.length;
            var range = input.createTextRange();
            input.value = c.name;
            range.moveStart("character", pos);
            range.select();
        }
    }
}
(continued)
```
Advantages of haXe/JS Development

A web developer with JavaScript experience can wonder why he should pass to program with haXe instead of continuing to develop directly with the language. There is more than one reason to explain why and all the reasons provided are per se very valid.

First and foremost it is the strictly typed environment. Having type checks at compile time reduces considerably the number of errors and typos. Be sure that this will not limit your programming style; type inference permits to declare types only when absolutely necessary; you have Dynamic to solve situations where many types are accepted and there is always the untyped backdoor.

The haXe compiler is smart and helps you to maintain your output file as lean as possible; in fact, you can have a huge library of classes but only the types used in your current application will be embedded in the output file; you are actually encouraged in building stable and solid libraries that can be reused from project to project. This is a notable feature because the programmer does not have to worry too much about maintaining short code and even more, he will not have to worry on how the JavaScript code has to be partitioned in files to include only when they are really necessary. The compiler produces just one file with just the needed code and no more.

The haXe-produced JavaScript plays nicely with other libraries. It does not change the native objects (the only exceptions are a couple of methods added to the Array type and the notation used to mark a type as a class) and, thus, does not conflict with third-party libraries. Using the extern keyword, an example is provided in Chapter 6 in the “JavaScript External Libraries” section. You can even use third-party libraries in your haXe code with a minimum effort.

The class and package system of haXe is very well integrated in JavaScript; both features are not available in current implementations of the most diffused browsers.

The JavaScript output is well-written and easy-to-read code; you can confidently build a library in haXe and distribute it to pure JavaScript developers without the fear that the code is hard to understand or debug.
Chapter 14: More Interactive Content with JavaScript

Summary

Programming JavaScript with haXe is a pleasure mainly for how haXe brings full OOP support to it. This chapter covered the potential of the environment and how to build useful applications. The following topics were discussed in detail:

- The kind of problems that JavaScript is commonly called to solve
- Firebug and other tools that can ease the development and debugging process of web applications
- The haXe JavaScript API
- How to validate a form on the client side
- How to create an auto-complete control that uses AJAX

In the next chapter, the communication layer is explored further introducing the haXe Remoting API. You learn how the communication between participants can become completely transparent using proxies.
Communication is everything in the development of web applications. The haXe environment provides not only ways to communicate between the client side and the server side, but also between different platforms on the same side. This chapter highlights the following topics:

- Remoting concepts
- Proxies
- Synchronous and asynchronous communications
- HTTP and socket connections
- Platform-to-platform communication

What Is haXe Remoting?

In this chapter, the client and server concepts assume a wider scope: The client is a piece of software that makes a request and the server is the software application that provides an answer. The physical location of client and server is not always relevant as the communication may happen on the same client PC or between two different PCs. The communication may involve the different platforms Flash, JavaScript, and Neko.

haXe Remoting is a library included in the haXe standard library that provides an easy and transparent way to communicate between clients and servers on different platforms. The communication is simplified because arguments and results are not limited to basic types but can be any kind of complex structures like enums, anonymous objects or class instances including arrays, lists, and hash tables. Furthermore, when proxy objects are used, the compiler can check for type correctness and coherent use of the remote methods available using Remoting.
Part II: Server Side, JavaScript, and Flash: Oh My!

**Remote Connection and Local Connection**

By using Remoting it is possible to communicate to a remote location over a network connection using the HTTP protocol or a socket connection, or to communicate locally between different components in the same context, usually a web page.

On the server side of a remote connection, you can use a Neko application or an Action Message Format (AMF) server, such as AMFPHP. The client side can involve Flash, JavaScript, or Neko.

When the communication is local you can have either JavaScript or Flash at the server or client side. A Flash-to-Flash communication is also meaningful and supported because it is not uncommon to have several Flash movies in the same context and the need to make them talk.

**Synchronous and Asynchronous Communications**

When a channel of communication is open and a request submitted, the client can act in one of two possible ways: It waits until a response is received or continues with the normal execution and triggers a proper method when the communication produces a result. The first is called synchronous communication because the flow of the messages is always alternating between a request and its answer. The second is defined asynchronous communication because a system does not have to wait for an answer before making a new request. When multiple requests are made, the sequence and timing of the responses cannot be predicted.

The biggest advantage of a synchronous connection is simplicity: The request method returns directly the response provided by the server. The biggest drawback is that the client code freezes until a response is received; this can be a negligible effect if the communication is local, but can be more an issue if the communication involves an HTTP connection. If an error occurs in the communication process, a standard exception is raised by the request method.

The asynchronous connection is a little more complex to set up because the request method must be invoked with an extra argument, a function that handles the response. The process does not lock the client at any step. Errors are handled by wiring a proper function to the `onError` function of the connection object. The `onError()` function is invoked each time an error occurs and the error message is passed as argument.

Sometimes it is not possible to choose between synchronous and asynchronous because the implementations are based on the native APIs that may not include every option.

**Pull or Push?**

A browser relies mainly on the HTTP protocol to exchange messages with the web servers. This is considered a pull type communication because the client pulls the information out of the server. The communication starts with the request and ends with the response. As soon as an answer is received, the connection is broken. For web pages and small downloads in general, this is an optimal solution because no resources are wasted in maintaining the channel of communication open when it is unpredictable if this channel will be reused or not.
The **push** technology works in a similar way because it is always the client that asks for a connection but instead of closing the connection immediately with the first response, it is kept open to allow further message exchanging. The nice thing about this kind of connection, more widely known as a **socket** connection, is that once it is established, the server side can send messages to the client without an explicit request as for the push label. This kind of connection is optimal in multiplayer games where multiple clients interact with a server and this must update the state of its clients as soon as new information is available. Most web browsers do not integrate any socket protocol, but Flash does with its **XMLSocket** class. haXe allows creating a socket communication directly from Flash or in JavaScript, too, using a small Flash movie as a bridge.

## The haXe Remoting API

Flash security policy imposes some limitations about how a Flash movie can interact with other movies or limit communications outside the current domain. To work around those limitations, you can set the proper Flash settings visiting this page [www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager04.html](http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager04.html) or adopting the appropriate security measures such as specifying the allowed domains in a socket connection.

The haXe Remoting API can be limited to support only certain versions of Flash because of limitations in the underlying APIs.

The package that contains the Remoting-related features is `haxe.remoting`. Table 15-1 describes all possible ways of communication in a local connection using the Remoting API. You will see that not all the possible combinations are covered. This is because some of them are not really useful in the development process and they have not been implemented. All the classes described are part of the `haxe.remoting` package.

<table>
<thead>
<tr>
<th>Client Platform</th>
<th>Server Platform</th>
<th>Static Class/Method to Create the Connection</th>
<th>Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash</td>
<td>Flash</td>
<td><code>FlashJsConnection.flashConnect()</code> (goes through JavaScript)</td>
<td>Yes</td>
</tr>
<tr>
<td>Flash</td>
<td>Flash</td>
<td><code>LocalConnection.connect()</code></td>
<td>Yes</td>
</tr>
<tr>
<td>Flash</td>
<td>JavaScript</td>
<td><code>Connection.jsConnect()</code></td>
<td>No</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Flash</td>
<td><code>Connection.flashConnect()</code></td>
<td>No</td>
</tr>
</tbody>
</table>

Table 15-2 shows the possible ways of communication in a remote connection using an HTTP or a socket connection.
Once they have been created, a Connection object or an AsyncConnection object always behave the same. All class peculiarities on the creation of a connection and their use will be discussed next.

The haXe Remoting is an open protocol and vendors are encouraged to develop alternative solutions. At the server side, the only working solution right now is developed with Neko and part of the haXe standard distribution but there are no reasons that impede to build alternatives in Java, PHP, or .Net.

### Table 15-2

<table>
<thead>
<tr>
<th>Client Platform</th>
<th>Server Platform</th>
<th>Static Class/Method to Create the Connection</th>
<th>Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash</td>
<td>AMF Server</td>
<td>AsyncConnection.amfConnect()</td>
<td>Yes</td>
</tr>
<tr>
<td>Flash</td>
<td>Socket Server</td>
<td>SocketConnection.socketConnect()</td>
<td>Yes</td>
</tr>
<tr>
<td>Flash</td>
<td>HTTP Server</td>
<td>AsyncConnection.urlConnect()</td>
<td>Yes</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Socket Server</td>
<td>SocketConnection.socketConnect()</td>
<td>Yes</td>
</tr>
<tr>
<td>JavaScript</td>
<td>HTTP Server</td>
<td>AsyncConnection.urlConnect</td>
<td>Yes</td>
</tr>
<tr>
<td>JavaScript</td>
<td>HTTP Server</td>
<td>Connection.urlConnect()</td>
<td>No</td>
</tr>
<tr>
<td>Neko</td>
<td>Socket Server</td>
<td>NekoSocketConnection.socketConnect()</td>
<td>No</td>
</tr>
<tr>
<td>Neko</td>
<td>HTTP Server</td>
<td>AsyncConnection.urlConnect()</td>
<td>Yes</td>
</tr>
<tr>
<td>Neko</td>
<td>HTTP Server</td>
<td>Connection.urlConnect()</td>
<td>No</td>
</tr>
</tbody>
</table>

Once they have been created a Connection object or an AsyncConnection object always behave the same. All class peculiarities on the creation of a connection and their use will be discussed next.

The haXe Remoting is an open protocol and vendors are encouraged to develop alternative solutions. At the server side, the only working solution right now is developed with Neko and part of the haXe standard distribution but there are no reasons that impede to build alternatives in Java, PHP, or .Net.

### Connection

A connection instance cannot be created using the class constructor but instead using one of the static `connect()` methods, as shown in the following example:

```javascript
var cnx = haxe.remoting.Connection.jsConnect();
var r = cnx.mypackage.Calc.sum.call([5, 4]);
```

The `cnx` object simply calls the `mypackage.Calc.sum()` method on the other side of the connection and returns the result.

To illustrate how it works, the `haxe.remoting.Connection` class implements Dynamic; this expedient is used by haXe to map a client invocation to a server method when the `call()` method is used. haXe will try to locate the `sum` static method inside the `Calc` class in the `mypackage` package contained in a compiled JavaScript script. The path used to locate a proper method is everything in between `cnx` and `call`. The `call()` method accepts an `Array<Dynamic>` of arguments that is passed to the server method and returns the result as a Dynamic value. If the method is used improperly the execution can fail silently or notify a runtime error to conform to the platform used. As described later in the “Proxy Objects” section, proxies can elegantly solve the problem of having untyped arguments and return value.
Chapter 15: Putting It All Together with haXe Remoting

By default in a local connection, the remote methods invoked must be static methods or methods of objects assigned to a static variable. In a remote connection, mainly for security reasons, the server must explicitly register the objects that you intend to render available to the client.

Table 15-3 describes the static creational methods available to generate a `Connection` instance from the `haxe.remoting.Connection` class. The returned type is always `Connection` and, thus, omitted.

<table>
<thead>
<tr>
<th>Connection Creational Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jsConnect(?name:String)</code></td>
<td>Used in a Flash movie to connect to a JavaScript host. The optional name parameter is used to direct the connection method invocations to a specific JavaScript object registered on the server using the bind static method.</td>
</tr>
<tr>
<td><code>flashConnect(objId:String)</code></td>
<td>Creates a connection to a Flash movie when the client is JavaScript. The argument is the id attribute of the HTML element that embeds the Flash Movie.</td>
</tr>
<tr>
<td><code>urlConnect(url:String)</code></td>
<td>Can be used in JavaScript or Neko to connect to a Neko HTTP server using a standard HTTP connection.</td>
</tr>
</tbody>
</table>

One important thing to note is that when JavaScript or Flash acts as server, it does not have to implement any specific code and it does not even need a `main()` method to start. The code must be simply compiled starting from a script that imports the `haxe.remoting.Connection` class. Of course the server code is really useful if there are methods and classes to invoke in there.

**Flash-to-JavaScript Connection**

An example of server-side code can be as simple as the following. The code works just fine in Flash or JavaScript when they are used as server in a local connection. The server code is compiled declaring just a class `Calc` with a single static method `sum` whose purpose is obvious.

```haxe
// content of the file Server.hx
import haxe.remoting.Connection;
class Calc
{
    public static function sum(x : Int, y : Int) : Int
    {
        return x + y;
    }
}
```
If the client is Flash, the code to access the JavaScript server can be the following. The client accesses the server code using the `cnx` instance.

```haxe
// content of the file Client.hx
class Client {
    public static function main() {
        var cnx = haxe.remoting.Connection.jsConnect();
        try {
            trace(cnx.Calc.sum.call([5,4]));
        } catch(e : Dynamic) {
            trace(Std.string(e));
        }
    }
}
```

Both scripts can be compiled at once saving the following text in a file `build.hxml`. Note that two client flash movies are created for both Flash 8 and 9.

```
- main Client
- swf out/client8.swf
- swf-header 320:40:20:ffffff
- swf-version 8
--next
- main Client
- swf out/client9.swf
- swf-header 320:40:20:ffffff
- swf-version 9
--next
Server
- js out/server.js
```

The compiled files can be produced executing the following command from the command line/console. Make sure you have created the `out` directory before the command execution.

```bash
> haxe build.hxml
```

You can finally produce an HTML page to host the example.

```html
<html>
<head><title>Flash-to-JS</title></head>
<script type="text/javascript" src="swfobject.js"></script>
<body>
<div class="window">
    <div class="label">Flash 8</div>
    <div id="flash8_container" class="cont">Flash 8</div>
</div>
<div class="window">
    <div class="label">Flash 9</div>
    <div id="flash9_container" class="cont">Flash 9</div>
</div>
</body>
</html>
```
Chapter 15: Putting It All Together with haXe Remoting

Flash movies are embedded using the SWFObject library that can be downloaded from this web address http://blog.deconcept.com/swfobject/

**Flash-to-JavaScript Connection Binding to an Object Instance**

When using JavaScript as a server, it is possible to bind a specific object instance to a connection instead of relying on the invocation of static methods. Reformulating the previous example to take advantage of the binding feature, you must rewrite the server script as follows. This time in the compilation command, it is necessary to use the -main switch to be sure that the binding occurs. The Connection.bind() static method requires a string that acts as a binding identifier and an object instance.

```haxe
// content of the file Server.hx
import haxe.remoting.Connection;

class Server
{
    public static function main()
    {
        Connection.bind("calc", new Calc());
    }
}

class Calc
{
    public function new() {} 
    public function sum(x : Int, y : Int) : Int
    {
        return x + y;
    }
}
```

On the client side, the connection must now be created using the binding name calc or else the class instance won’t be accessible. The try/catch block has been removed to simplify the readability.

```haxe
// content of the file Client.hx
class Client
{
    public static function main()
    {
        var calc = haxe.remoting.Connection.jsConnect("calc");
        trace(calc.sum.call([5,4]));
    }
}
```
**JavaScript-to-Flash Connection**

Reversing the Flash-to-JavaScript example using Flash as server and JavaScript as client does not present any particular inconvenience; the only thing to take care of is the timing of the communication. An embedded Flash movie always presents a certain time delay before being available for use. You can circumvent the problem by delaying the execution 500 milliseconds or so or by using the JavaScript DOM functionalities to monitor the presence and availability of the Flash movie. The server script is the same as the previous code and the client can be as follows.

```haxe
// content of the file Client.hx
class Client
{
    public static function main()
    {
        haxe.Timer.delayed(function()
        {
            var cnx = haxe.remoting.Connection.flashConnect("flashmovie8");
            trace(cnx.Calc.sum.call([5,4]));
        }, 500);
    }
}
```

The method `flashConnect()` looks for an HTML element with id equal to `flashmovie8` that embeds the Flash server movie.

**AsyncConnection**

The `haxe.remoting.AsyncConnection` class works almost the same as `Connection`. The biggest difference is in the use of the `call()` method. The method accepts an extra optional argument that is the function invoked when the client receives a response from the server. If omitted the client sends a request to the server but it is not possible to intercept the answer. This can be useful when the remote method returns `Void`.

Table 15-4 describes the static creational methods used to generate an `AsyncConnection` instance from the `haxe.remoting.AsyncConnection` class. The returned type is always `AsyncConnection` and, thus, omitted.

**Table 15-4**

<table>
<thead>
<tr>
<th>AsyncConnection Creational Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>urlConnect(url:String)</code></td>
<td>Used on any platform to connect to a Neko HTTP server. The argument is the URL of the server.</td>
</tr>
<tr>
<td><code>amfConnect(gatewayUrl:String)</code></td>
<td>Used in Flash to connect to an AMF Server. The argument is the URL of the gateway service.</td>
</tr>
</tbody>
</table>

Both methods create connection for a remote kind of communication.
Chapter 15: Putting It All Together with haXe Remoting

JavaScript-to-Neko HTTP Server

To work with an asynchronous connection the client code can contain the optional data handling function and it is advised to always wire an onError method to intercept communication errors. The following code describes a connection from a JavaScript client code and a Neko HTTP server.

```haxe
// content of client.hx
import haxe.remoting.AsyncConnection;
class Client
{
    public static function main()
    {
        var cnx = AsyncConnection.urlConnect("http://localhost:2000/server.n");
        cnx.onError = function(e)
        {
            trace("Error: " + Std.string(e));
        }
        cnx.calc.sum.call([5,4], function(data)
        {
            trace(data);
        });
    }
}

// content of server.hx
class Server
{
    public static function main()
    {
        var server = new neko.net.RemotingServer();
        server.addObject("calc", new Calc());
        if(server.handleRequest())
            return;
        // handle other requests (not remoting)
        neko.Lib.print("Neko remoting server waiting for a request");
    }
}
class Calc
{
    public function new() { }
    public function sum(x : Int, y : Int) : Int
    {
        return x + y;
    }
}
```

The RemotingServer class is described later in this chapter in the “Proxy Objects” section and in the “TicTacToe Game” section. The most important thing is that a client cannot invoke any method available on the server but those belonging to the instances registered with the addObject() method. More than one object can be registered on the server. The handleRequest() method detects if the request comes from a Remoting Connection object and if this is the case, it tries to invoke the corresponding method. If the request is not of this kind, the method returns true and eventually the server script can try to handle the request differently. This mechanism allows reusing of the same script to deal with Remoting requests and standard page requests in the same script and producing just one Neko file.
Part II: Server Side, JavaScript, and Flash: Oh My!

**Flash to AMF Server**

An Action Message Format Server (AMF server) is a kind of server that implements the Adobe protocol of communication for Flash movies. When connecting to such a server, Flash can send and receive ActionScript objects and types. Because the Neko HTTP server is a superior solution, haXe does not provide a server implementation of AMF but provides the code to implement the client-side part of the communication. It is self-evident that some types like enums are not available in this context because an AMF server cannot support them.

In the example, the AMFPHP server is used on the server side. AMFPHP is an open source implementation for PHP realized originally by Wolfgang Hamann and released under the GNU License (GNU is a recursive acronym for GNU is Not Unix). The code has been tested with both versions 1.2.6 and 1.9 beta. To try the following code, be sure you have installed PHP correctly (PHP 5 is required for version 1.9) and mapped the AMFPHP folder properly to a path such as the one used in the code. The client code is very easy and doesn’t differ from the ones seen earlier.

```haxe
// content of the file Client.hx
import haxe.remoting.AsyncConnection;
class Client
{
    public static function main()
    {
        var cnx = AsyncConnection.amfConnect("http://localhost/amf/gateway.php");
cnx.onError = function(e)
        {
            trace("Error: " + Std.string(e));
        }
cnx.Calc.sum.call([5,4], function(data)
        {
            trace(Std.string(data));
        });
    }
}
```

For the server side, create a file Calc.php in the services folder of the AMFPHP installation directory. The file must contain the following PHP code:

```php
<?php
class Calc {
    function Calc() {
        $this->methodTable = array (
            "sum" => array (
                "access" => "remote",
                "description" => "Executes a sum"));
    }
    function sum($x, $y) {
        return $x +$y;
    }
}
?>
```
Chapter 15: Putting It All Together with haXe Remoting

**FlashJsConnection**

The `haxe.remoting.FlashJsConnection` class connects two Flash movies using JavaScript as a bridge Table 15-5. The client movie identifies the server movie using the `id` attribute of the HTML element that is used to embed it.

**Table 15-5**

<table>
<thead>
<tr>
<th>FlashJsConnection Creational Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flashConnect(objId:String)</code></td>
<td>Used to create a connection between two Flash movies inside a web page context. The <code>objId</code> argument is a string identifier that corresponds to the <code>id</code> attribute of the HTML element that embeds the Flash movie that acts as server.</td>
</tr>
</tbody>
</table>

To make a `FlashJsConnection` work, you must also provide a JavaScript file compiled with haXe that will act as a bridge. Such a file can be an `.hx` file containing just the following line of code:

```javascript
import haxe.remoting.FlashJsConnection;
```

**LocalConnection**

The `haxe.remoting.LocalConnection` class relies on the Flash LocalConnection API implementation. This channel of communication permits the exchange of messages between different Flash movie instances that may also be located in different contexts (not on the same web pages). When you use `LocalConnection`, it is possible to open a bridge between a Flash movie executed in the Flash Player and a movie embedded in a page Table 15-6.

**Table 15-6**

<table>
<thead>
<tr>
<th>LocalConnection Creational Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connect(name:String,</code>&lt;br&gt;<code>?allowDomains:Array&lt;String&gt;)</code></td>
<td>Used to create a connection between two Flash movies. The first argument identifies the connection by name. If the communication must happen in Flash movies received from different web domains, those domains must be specified in the optional <code>allowDomains</code> argument.</td>
</tr>
</tbody>
</table>

It is possible to close the communication prematurely using the `closeConnection()` method.

*LocalConnection has not been very well implemented and can be quite slow. If you don’t have to work with Flash prior to version 8, FlashJsConnection is always a better choice.*
SocketConnection

As already explained, a socket connection is a kind of communication that remains open between a client and a server and that must be actively closed by one of the sides Table 15-7. To create this kind of connection, the haxe.remoting.SocketConnection class is used. Note that this class is used both at the server side in a Neko application and at the client side, when the client is Flash or JavaScript.

Table 15-7

<table>
<thead>
<tr>
<th>SocketConnection Creational Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>socketConnect(s:Socket)</td>
<td>Creates a SocketConnection object when the client is Flash or JavaScript. The argument is a Socket object: an instance of flash.net.XMLSocket for Flash 9, flash.XMLSocket for Flash 8, or js.XMLSocket for JavaScript.</td>
</tr>
</tbody>
</table>

JavaScript has no native implementation for socket communication. This limit can be overcome using a tiny Flash movie as a bridge. The Flash movie can be built by just importing the SocketWrapper class that adds the bridging features. JavaScript then uses the movie to communicate with the server.

The Flash movie can be compiled just with this single line:

```javaScript
import haxe.remoting.SocketWrapper;
```

A full and more advanced example of Socket connection is discussed later in this chapter in the “TicTacToe Game” section.

NekoSocketConnection

A haxe.remoting.NekoSocketConnection is used to connect a client Neko application to a Neko socket server Table 15-8.

Table 15-8

<table>
<thead>
<tr>
<th>NekoSocketConnection Creational Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>socketConnect(s:neko.net.Socket, ?r:neko.net.RemotingServer )</td>
<td>Creates a connection to a Neko socket server from a Neko application. The first argument is Socket used for the communication. The second optional argument is used when an incoming request is triggered by the server.</td>
</tr>
</tbody>
</table>
Chapter 15: Putting It All Together with haXe Remoting

A Neko-to-Neko socket connection can be useful for many purposes such as synchronizing the database contents of different servers or exchanging files. Remember that Neko is not only used on the server but can also work on a client PC as a standalone application; refer to Chapters 17 to 19 to read more about the subject.

**AsyncDebugConnection, DelayedConnection, and AsyncAdapter**

The `haxe.remoting.AsyncDebugConnection` and the `DelayedConnection` wrappers add some useful features to another connection. The `AsyncDebugConnection` accepts an `AsyncConnection` object as an argument of its constructor. Each time a message exchange is executed, the class simply sends to the `trace()` function the name of the function requested and the arguments passed to it.

A `DelayedConnection` object is created using the static method `create()` that has no arguments. The object created is a stub used to accumulate the calls to the server; as soon as a real connection is associated to the `DelayedConnection` instance using the `connection` variable, the calls are performed. Any subsequent use of `call()` is executed immediately.

The `AsyncAdapter` is a wrapper class that builds an asynchronous connection over an existing synchronous connection. It can be useful when the asynchronous connection required, is not implemented. This adapter allows implementing the client once using the asynchronous API and use of any of the possible ways of communication.

**SocketProtocol**

The `haxe.remoting.SocketProtocol` class is used by the `SocketConnection` class to send and receive messages to and from the server and to transform their content into haXe structures using the `haxe.Serializer` and `haxe.Unserializer` classes. Unless you need to extend the socket functionalities, you do not have to deal directly with this class. In any case you probably want to import the `SocketProtocol` in your script because it contains the `haxe.remoting.Socket typedef` that is a handy shortcut type definition to the platform-specific socket classes.

**Proxy Objects**

The `call()` method of the many connection objects available permits the easy exchange of messages with a server. The path of the method required and the types of arguments and the results are not checked by the compiler because the connection objects implement `Dynamic`. haXe resolves this issue introducing two special classes: `Proxy` and `AsyncProxy` used respectively in synchronous and asynchronous connections.

A `Proxy` class is created extending the `Proxy<ServerClassToInvoke>` type. This is enough to activate the haXe magic. The created proxy class acts in the client code and is checked by the compiler as if it was an instance of `ServerClassToInvoke`. 
In the following code, the explored Calc example can be changed to take advantage of the proxy API. This time the connection is synchronous from JavaScript to a remote Neko HTTP server. Only the client code is reported.

```haxe
import haxe.remoting.Connection;
class CalcProxy extends haxe.remoting.Proxy<Calc> {}
class Client {
    public static function main() {
        var cnx = Connection.urlConnect("http://localhost:2000/server.n");
        var calc = new CalcProxy(cnx.Calc);
        trace(calc.sum(5, 4));
    }
}
```

The `CalcProxy` class is defined in just one line of code. Every `Proxy` instance must be created passing to the constructor the correct path to the remote object that must be associated to the proxy. A proxy instance behaves exactly as the mirrored class.

The methods of an `AsyncProxy` act in a similar way but accept an optional extra argument that is the function that is delegated to handle the result when this becomes available. To illustrate the use of AsyncProxy and to resume what has been discussed so far, here is a more complex and real example.

On the server side is a database of persons. Those persons can have parents and children referenced using a unique numeric identifier. The `Person` object also keeps record of the year of birth. To alleviate the sample code, the `PersonDatabase` class is hard coded with its example data, but the code can be easily adapted to load such data from a real database using SPOD or a direct SQL connection.

On the client side a JavaScript interface is implemented using an AJAX technique. Every time a link to a person in the page is selected, the data of this person is loaded from the server and applied on the page without the need to reload the entire page.

The `Person` class is very simple and it is used as a specialized data container, as shown in the following example:

```haxe
typedef PersonId = {
    id : Int,
    name : String
} 
class Person {
    public var id(default, null) : Int;
    public var name : String;
    public var birthyear : Int;
    public var father : Null<PersonId>;
    public var mother : Null<PersonId>;
```
public var children(default, null) : List<PersonId>;
public function new(id : Int, name : String, birthyear : Int)
{
    children = new List();
    this.id = id;
    this.name = name;
    this.birthyear = birthyear;
}
public function getId()
{
    return { id : id, name : name };
}
}

The PersonDatabase class contains the data and has two methods to retrieve them:
a listIds() method that returns an array containing the id and the name of the corresponding person, and getById() that returns an instance of Person with all the data available about the selected id.

import Person;
class PersonDatabase
{
    private var persons : Array<Person>;
    public function new()
    {
        var rs = new Person(1, "Richard Shakespeare", 1561);
        var js = new Person(2, "John Shakespeare", 1601);
        rs.children.push(js.getId());
        js.father = rs.getId();
        // ... others omitted for brevity
        persons = [rs, js /* add more here */];
    }
    public function listIds() : Array<PersonId>
    {
        var ids = new Array();
        for(person in persons)
            ids.push({ id : person.id, name : person.name });
        ids.sort(function(x, y){
            return if(x.name < y.name) -1 else if(x.name > y.name) 1 else 0; });
        return ids;
    }
    public function getById(id : Int) : Null<Person>
    {
        for(person in persons)
            if(person.id == id)
                return person;
        return null;
    }
}
Part II: Server Side, JavaScript, and Flash: Oh My!

The Server class to implement the HTTP Server is very simple. An instance of the PersonDatabase is registered on the RemotingServer using the addObject method.

```haxe
class Server {
    public static function main() {
        var server = new neko.net.RemotingServer();
        server.addObject("PersonDatabase", new PersonDatabase());
        if(server.handleRequest())
            return;
        // handle other requests (not remoting)
        neko.Lib.print("Neko remoting server waiting for a request");
    }
}
```

Although the objects must be explicitly registered to be used by a client, their methods are all available. It is possible to restrict the access to certain methods using the setPrivatePrefix() on the RemotingServer() instance. All methods starting with the specified prefix cannot be directly invoked by a client.

The client code uses an AsyncConnection to communicate with the server. First the whole list of persons is loaded using listIds(). The method has a reference to the fillList() function as argument. The fillList() function iterates over the received array and builds an HTML list of elements. Each element is enhanced with an onclick event that once triggered uses the db object to load all the data about the selected person. Parents and children are also active links to refresh the page content.

```haxe
import js.Dom;
import Person;

class PersonDatabaseProxy extends haxe.remoting.AsyncProxy<PersonDatabase> {}

class Client {
    public static function main() {
        new Client("http://localhost:2000/server.n");
    }
    public function new(url) {
        var cnx = haxe.remoting.AsyncConnection.urlConnect(url);
        cnx.onError = function(e) {
            trace("Error: " + Std.string(e));
        }
        db = new PersonDatabaseProxy(cnx.PersonDatabase);
        db.listIds(fillList);
    }
    private var db : PersonDatabaseProxy;
    private function fillList(ids : Array<PersonId>) {
        var ul = js.Lib.document.getElementById("persons");
        ul.innerHTML = '';
        for(id in ids)
```
Chapter 15: Putting It All Together with haXe Remoting

```javascript
{  
    var li = js.Lib.document.createElement("li");  
    li.appendChild(createLink(id));  
    ul.appendChild(li);  
}

private function fillTemplate(person : Person)  
{  
    replaceContent("name", [createLabel(person.name)]);  
    replaceContent("birthyear", [createLabel(Std.string(person.birthyear))]);  
    replaceContent("father", [createLink(person.father)]);  
    replaceContent("mother", [createLink(person.mother)]);  
    var nodes = new Array();  
    for(child in person.children)  
        {  
            if(person.children.first() != child)  
                nodes.push(untyped js.Lib.document.createTextNode("", "));  
            nodes.push(createLink(child));  
        }  
    replaceContent("children", nodes);  
}

private function replaceContent(id, children : Array<HtmlDom>)  
{  
    var el = js.Lib.document.getElementById(id);  
    el.innerHTML = '';  
    for(child in children)  
        el.appendChild(child);  
}

private function createLabel(v)  
{  
    var s = js.Lib.document.createElement("span");  
    s.innerHTML = v;  
    return s;  
}

private function createLink(pid)  
{  
    if(null != pid)  
    {  
        var a = js.Lib.document.createElement("a");  
        a.innerHTML = pid.name;  
        var self = this;  
        untyped  
        {  
            a.href = "#";  
            a.onclick = function(e)  
            {  
                self.db.getById(pid.id, self.fillTemplate);  
                return false;  
            }  
        }  
        return a;  
    }  
    else  
        return createLabel("<em>unknown</em>");  
}
}
Part II: Server Side, JavaScript, and Flash: Oh My!

The preceding example, embedded in the HTML page shown in the following code, should produce an output as the one depicted in Figure 15-1. The page displayed has some basic CSS rules applied that have been omitted for brevity.

```html
<html>
<head>
<title>Persons</title>
<link href="style.css" rel="stylesheet" type="text/css" />
</head>
<body>
<ul id="persons">
</ul>
<div id="person">

<div>
  name: <strong> <span id="name" >- </span> </strong>
</div>
<div>
  birth year: <span id="birthyear" >- </span>
</div>
<div>
  father: <span id="father" >- </span>
</div>
<div>
  mother: <span id="mother" >- </span>
</div>
<div>
  children: <span id="children" >- </span>
</div>
</div>
<div id="haxe:trace"></div>
<script type="text/javascript" src="client.js"></script>
</body>
</html>
```

Sometimes in Proxy<Class>, the Class cannot or must not be compiled in the client output. It is then needed to use conditional compilation to avoid misuses; an alternative solution is to provide an Interface that can be implemented by the Class; the Proxy<Interface> can use it instead of the concrete Class.

Figure 15-1

- Anne Hathaway
- Anne Shakespeare
- Edmund Shakespeare
- Elizabeth Hall
- Gilbert Shakespeare
- Hamnet Shakespeare
- Joan Shakespeare
- Joan Shakespeare
- John Hall, Dr.
- John Shakespeare
- Judith Shakespeare
- Margaret Shakespeare
- Mary Arden
- Richard Quiney Vintner
- Richard Shakespeare
- Richard Shakespeare
- Robert Arden
- Shaksper Quiney Vintner
- Susanna Shakespeare
- Thomas Quiney Vintner
- Thomas Quiney Vintner
- William Shakespeare

name: William Shakespeare
birth year: 1564
father: John Shakespeare
mother: Mary Arden
children: Judith Shakespeare, Hamnet Shakespeare, Susanna Shakespeare
Chapter 15: Putting It All Together with haXe Remoting

TicTacToe Game

In the SocketConnection description, it has been said that this kind of connection is ideal in the realization of multiplayer games. To conclude this chapter with a practical example of multiuser interaction, a Flash client and Neko server are built to play the famous TicTacToe game. Figure 15-2 illustrates a possible interaction between a Flash movie embedded in a web page and one executed as a standalone player. Notice that to allow the communication between a local Flash movie and a remote server, it is necessary to change the Flash Player security settings explicitly selecting the movie that can perform the remote calls.

![TicTacToe Game](image)

Figure 15-2

It's your turn.

Wait for the adversary move.
The files that compose the project are organized as described in Figure 15-3.

![Image of file structure]

**Figure 15-3**

The Flash movie is created starting from a `GameClient` instance. It is built on Flash version 8 but can be easily adapted to Flash 9 if needed. One purpose of this class is to visualize the current status of the game grid and to show to the user the messages transmitted by the server. All the text messages have been encapsulated in this class. Having the text messages all in the same place is a good practice, particularly useful when the developer is requested to produce a localized version of the game. The other purpose of the class is to communicate the user choices to the server. This is done using an `AsyncProxy` based on the `IPlayerServer` interface. Instead of using directly the `PlayerServer` to create a proxy, an interface has been extracted with just the methods that can be invoked by the Flash movie interface. The `PlayerServer` method `_turn` that can be executed only on request of some methods on the server side is protected by a direct interaction with the client using the underscore (_) prefix registered using the `setPrivatePrefix()` method. The `PlayerServer` class represents an individual player on the server. Every time a new player connects to the server, a new instance of this class is created and associated to the current connection. The player is then added to an attendants queue. On each change of the queue the server tries to associate two nonplaying players to start a new game session. A `GameBoard` instance is created and associated to both players. The class contains the logic and mechanisms of a TicTacToe game. The `PlayerServer` interacts with this object to check if any of the players won or if the match ended in a tie. The match can also end prematurely if one of the players disconnects in the middle of the competition; in this case the server immediately notifies the other player. The `PlayerServer` communicates with the client using a proxy built upon the interface `IGameClient`. The user interface has been created to limit the possible interactions between the server and the client to the bare minimum required; the server is not interested, and should not be, in how the client is implemented and just needs to access its more generic methods.

The `IGameClient` interface has some methods that are only used to notify a message to the user and others that also update the grid. The entire grid is passed as argument in the form of an array of arrays of integer values. Each value can assume the value 0 meaning that the cell is not occupied by any symbol, the value 1 to assign the cell to player one and -1 for player two.

```typescript
interface IGameClient
{
    function waitAdversary() : Void;
    function adversaryQuit() : Void;
    function tie(g : Array<Array<Int>> ) : Void;
    function winner(g : Array<Array<Int>> ) : Void;
    function looser(g : Array<Array<Int>> ) : Void;
    function invalidMove() : Void;
    function yourTurn(g : Array<Array<Int>> ) : Void;
    function otherTurn(g : Array<Array<Int>> ) : Void;
}
```
The IPlayerServer interface is very short; the attend() method is invoked by the user interface when the user expresses the desire to join in a new game, whereas the second method place() is used to set a symbol at the specified row and column. Note that the place() method is executed each time any of the two players clicks on the grid at any time, and also when it is not their turn. The action is validated and simply ignored by the server if inappropriate.

interface IPlayerServer
{
    function attend() : Void;
    function place(r : Int, c : Int) : Void;
}

The GameBoard class maintains the playing grid that is defined by three triplets of 0 values. The class also maintains the results of the game to determine if one player won or the game was concluded in a tie. The system adopted to maintain the result is weirdly brainiac but it is efficient and short in lines of code. Each position of the results array can contain a number between −3 and 3. If the number is equal to 3 the first player wins; if the number is equal to −3 it is the second player who wins. Every value in the middle is not relevant. The positions from 0 to 5 are for the sum of each row and each column. The positions 8 and 13 are for the two diagonals. Every other position in the array can be potentially used by the algorithm but can never reach a winning value.

import haxe.remoting.SocketConnection;
class GameBoard
{
    public var grid(default, null) : Array<Array<Int>>;
    private var results : Array<Int>;
    private var moves : Int;
    public function new()
    {
        grid = [[0, 0, 0],[0, 0, 0],[0, 0, 0]];
        results = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
        moves = 0;
    }
    public function place(isPlayerOne : Bool, row : Int, col : Int) : Bool
    {
        if( grid[row][col] != 0 )
            return false;
        var point = if(isPlayerOne) 1 else -1;
        grid[row][col] = point;
        results[row] += point;        // cumulate points for rows
        results[3+col] += point;      // cumulate points for cols
        results[8+row-col] += point;  // cumulate points for first diagonal
        results[15-row-col] += point; // cumulate points for second diagonal
        moves++;
        return true;
    }
    public function checkTie() : Bool
    {
        return moves == 9;
    }

    (continued)
Part II: Server Side, JavaScript, and Flash: Oh My!

(continued)

    public function checkWinner() : Bool
    {
        for(r in results)
            if(r == 3 || r == -3)
                return true;
        return false;
    }
}

In the GameClient.main() method a new socket is created and associated to a SocketConnection. A new instance of the class is also created and stored in a static variable client. The client is also registered in the _global space of the Flash environment to be reachable when a message is received from the server.

The socket connects to a localhost address on port 2001 but any domain and port can be used as long as it is not used by another process.

The class has a very simple logic all dedicated to invoke the remote methods and to draw the user interface.

    import haxe.remoting.SocketProtocol;
    class GameApi extends haxe.remoting.AsyncProxy<IPlayerServer> {}
    class GameClient implements IGameClient
    {
        public static var client : GameClient;
        public static function main()
        {
            var socket = new haxe.remoting.Socket();
            socket.onConnect = onConnect;
            var cnx = haxe.remoting.SocketConnection.socketConnect(socket);
            client = new GameClient(new GameApi(cnx.PlayerServer));
            flash.Lib._global.client = client;
            socket.connect("localhost", 2001);
        }
        public static function onConnect(connected)
        {
            if(!connected)
                client.message("Unable to connect");
        }
        private var server : GameApi;
        private var ended : Bool;
        public function new(server)
        {
            this.server = server;
        }
        public function yourTurn(g : Array<Array<Int>>)
        {
            ended = false;
            grid(g);
            message("It's your turn.");
        }
Chapter 15: Putting It All Together with haXe Remoting

public function otherTurn(g : Array<Array<Int>>)
{
    ended = false;
    grid(g);
    message("Wait for the adversary move.");
}

public function waitAdversary()
{
    message("Wait for another player.");
}

public function tie(g : Array<Array<Int>>)
{
    ended = true;
    grid(g);
    message("It's a tie! Click the grid to find another player.");
}

public function winner(g : Array<Array<Int>>)
{
    ended = true;
    grid(g);
    message("You win! Click the grid to find a new adversary.");
}

public function looser(g : Array<Array<Int>>)
{
    ended = true;
    grid(g);
    message("You Lose! Click the grid to find a new adversary.");
}

public function invalidMove()
{
    message("Invalid move ... ");
}

public function adversaryQuit()
{
    ended = true;
    message("Your adversary has quit! Click the grid to find another player.");
}

private function drawCross(r, c)
{
    board.moveTo(side*c+offset*2, side*r+offset*2);
    board.lineTo(side*(c+1), side*(r+1));
    board.moveTo(side*(c+1), side*r+offset*2);
    board.lineTo(side*c+offset*2, side*(r+1));
}

private function drawCircle(row, col)
{
    var r = (side-offset*2)/2;
    var x = side*col+offset+side/2;
    var y = side*row+offset+side/2;
    var t = Math.tan(Math.PI/8);
    var s = Math.sin(Math.PI/4);
}
(continued)

```actionscript
board.moveTo(x+r, y);
board.curveTo(r+x, t*r+y, s*r+x, s*r+y);
board.curveTo(t*r+x, r+y, x, r+y);
board.curveTo(-t*r+x, r+y, -s*r+x, s*r+y);
board.curveTo(-r+x, t*r+y, -r+x, y);
board.curveTo(-r+x, -t*r+y, -s*r+x, -s*r+y);
board.curveTo(-t*r+x, -r+y, x, -r+y);
board.curveTo(t*r+x, -r+y, s*r+x, -s*r+y);
board.curveTo(r+x, -t*r+y, r+x, y);
}
```

```actionscript
private var board : flash.MovieClip;
public function grid(g : Array<Array<Int>> ) : Void
{
    if(board == null)
    {
        board = flash.Lib.current.createEmptyMovieClip("board", 1);
        board.onMouseUp = onMouseUp;
    }
    board.clear();
    board.lineStyle(linewidth, linecolor, 100);
    board.moveTo(offset+side, offset);
    board.lineTo(offset+side, offset+side*3);
    board.moveTo(offset+side*2, offset);
    board.lineTo(offset+side*2, offset+side*3);
    board.moveTo(offset, offset+side);
    board.lineTo(offset+side*3, offset+side);
    board.moveTo(offset, offset+side*2);
    board.lineTo(offset+side*3, offset+side*2);
    for(r in 0...3)
    for(c in 0...3)
        switch(g[r][c])
        {
            case 1: // player one
                drawCross(r, c);
            case -1: // player two
                drawCircle(r, c);
            default:
                continue;
        }
}
```

```actionscript
private var tf : flash.TextField;
public function message(s : String) : Void
{
    if(tf == null)
    {
        tf = flash.Lib.current.createTextField("tf", 2, 0,
                side*3+offset*2, side*3+offset*2, fontsize*2+12);
        tf.multiline = tf.wordWrap = true;
        tf.setNewTextFormat(new flash.TextFormat(fontname, fontsize,
                fontcolor, true, false, false, null, null, "center"));
    }
    tf.text = s;
}
```
Chapter 15: Putting It All Together with haXe Remoting

```haxe
private function onMouseUp()
{
    if(ended)
        server.attend();
    else
        placeSymbol();
}

private function placeSymbol()
{
    var x = board._xmouse;
    var y = board._ymouse;
    for(r in 0...3)
    {
        if(y < offset || y > (side*(r+1)+offset))
            continue;
        for(c in 0...3)
        {
            if(x > offset && x < (side*(c+1)+offset))
            {
                server.place(r, c);
                return;
            }
        }
    }

private static var offset = 20;
private static var side = 120;
private static var fontname = "Arial";
private static var fontsize = 20;
private static var fontcolor = 0x006699;
private static var linewidth = 10;
private static var linecolor = 0xcc6600;
```
(continued)

```haxe
public function _turn() : Void
{
    isActive = true;
    other.isActive = false;
    client.yourTurn(board.grid);
    other.client.otherTurn(board.grid);
}
public function place(row : Int, col : Int) : Void
{
    if(board == null || !isActive)
        return;
    if(board.place(isPlayerOne, row, col))
    {
        if(board.checkWinner())
        {
            client.winner(board.grid);
            other.client.looser(board.grid);
            TicTacToeServer.clear(this);
        } else if(board.checkTie()) {
            client.tie(board.grid);
            other.client.tie(board.grid);
            TicTacToeServer.clear(this);
        } else
            other._turn();
    } else
        client.invalidMove();
}
public function attend() : Void
{
    if(!TicTacToeServer.isAttendant(this))
        TicTacToeServer.addAttendant(this);
}
```

The `TicTacToeServer` is the entry class on the server side. A `ThreadRemotingServer` is instantiated in the `main()` method and wired to the `initClientApi` and `clientDisconnected` methods. The first is triggered each time a new player connects to the server, the second when a player disconnects from the server. The `ThreadRemotingServer` permits the simultaneous connection of many users. The management of multiple connections is very efficient and only limited by the server hardware. The class also manages the beginning and conclusion of each game session. Each time a new client connects and the corresponding player has been added to the attendants list, the class creates a game session for each couple of players.

```haxe
import haxe.remoting.SocketConnection;
import neko.net.RemotingServer;
class TicTacToeServer
{
    private static function main()
    {
        var s = new neko.net.ThreadRemotingServer();
        s.initClientApi = initClientApi;
        s.clientDisconnected = clientDisconnected;
    }
```
Chapter 15: Putting It All Together with haXe Remoting

```java
trace("TicTacToe server...");
s.run("localhost", 2001);
}

private static var attendants = new List<PlayerServer>();

private static function initClientApi(scnx:SocketConnection, server:RemotingServer)
{
    var p = new PlayerServer(scnx);
    server.setPrivatePrefix("_");
    server.addObject("PlayerServer", p);
    (cast scnx).__player = p;
    addAttendant(p);
}

public static function clear(p : PlayerServer)
{
    if(p.board == null)
        return;
    p.board = p.other.board = null;
    p.other = p.other.other = null;
}

private static function clientDisconnected(scnx)
{
    var p : PlayerServer = (cast scnx).__player;
    if(p.other != null)
        p.other.client.adversaryQuit();
    clear(p);
    manageGames();
}

private static function manageGames()
{
    while(attendants.length > 1)
    {
        var p1 = attendants.pop();
        var p2 = attendants.pop();
        p1.isPlayerOne = true;
        p2.isPlayerOne = false;
        p1.board = p2.board = new GameBoard();
        p1.other = p2;
        p2.other = p1;
        p1._turn();
    }
    if(attendants.length > 0)
        attendants.first().client.waitAdversary();
}

public static function addAttendant(p)
{
    attendants.add(p);
    manageGames();
}

public static function isAttendant(p)
{
    return Lambda.has(attendants, p);
}
```
Part II: Server Side, JavaScript, and Flash: Oh My!

The client and the server can be compiled using the build.hxml file.

```
-main GameClient
-swf out/client.swf
-swf-version 8
--flash-strict
-swf-header 400:452:20:ffffff
--next
-main TicTacToeServer
-neko out/server.n
```

To start the server execution, use the following command from the same directory of server.n:

```
> neko server.n
```

You can embed the user interface in a web page that can be executed locally if the security settings have been adjusted properly or by using the nekotools server or the Apache web server. In any case make sure that your game server and web server run on different ports or one of the two will fail.

Summary

In this chapter all possible ways of communication using the Remoting API were discussed. After some theory about local and remote connections, synchronous and asynchronous communications, and pull and push technologies, the haXe Remoting API was described highlighting the following possibilities:

- Flash-to-Flash communication (using LocalConnection or JavaScript)
- Flash-to-JavaScript and reverse
- Flash/JavaScript/Neko-to-Neko HTTP server
- Flash/JavaScript/Neko-to-Neko socket server
- Flash to AMF server

The next chapter focuses on advanced features existing in the haXe standard libraries such as XML fast access and validation, Reflection, RTTI and haXe magic.
Part III
Extending the Possibilities

Chapter 16: haXe Advanced Topics
Chapter 17: Desktop Applications with Neko
Chapter 18: Desktop Flash using SWHX
Chapter 19: Multimedia with Neko
Chapter 20: Extending haXe with C/C++
The chapter is dedicated to the advanced classes in the haXe package. You will learn how to stretch the bounds of programming with haXe and how to achieve results otherwise impossible to obtain. The following topics are covered:

- How to use advanced XML handling
- What reflection API is
- How to use Runtime Type Information
- Serialization and deserialization
- haXe Magic

**XML**

The base XML class is good because it provides you with a unified way to deal with XML documents in a cross-platform way but it is sometimes a little too repetitive to use. The haxe.xml package contains two classes that can really speed your coding: Fast for fast retrieval of elements and Check to rapidly create a validation schema for your documents.

**Fast**

The Fast class is a cool class to access and traverse XML documents effectively and with a less verbose approach than the one provided by the standard XML class. It has been developed to rapidly retrieve nodes of the Element type. Fast instance variables have a peculiar characteristic: They are not used to store and retrieve data directly but they are used as operators to query an Xml node. Each variable performs a different kind of filtering action on the node. If the filter needs an
Part III: Extending the Possibilities

argument to be applied, this is passed as an instance variable to the operator. Take a look at the following example:

```haxe
class Main {
    public static function main()
    {
        var doc = Xml.parse('<person name="John"/>
        var n = new haxe.xml.Fast(doc);
        trace(n.node.person.att.name);
    }
}
```

The `n` variable is an `Xml` node wrapped in a `Fast` object; the node variable states that the first element whose name is equal to `person` must be returned. The type of `person` is `Fast` to permit further filtering and traversing; thus it is possible to access the attribute name using `att.name`.

The `Xml` class returns a null value when an inexistent node or attribute is retrieved while the `Fast` class throws an exception when this happens. This runtime check ensures that names are used correctly to query the XML document.

**Traversing Elements**

An XML node can be traversed using the following operators: `elements`, `nodes`, and `node`. The first returns all the elements that are children of the current node. The `nodes` operator does the same but filters the nodes by name, and `node` returns only the first node of the list.

Suppose you have an XML document with the following content:

```xml
<?xml version="1.0" ?>
<people>
    <person>
        <name>John Doe</name>
        <bio>
            Biography goes here ...
        </bio>
    </person>
    <author>
        <name>William Shakespeare</name>
        <bio>
            Very long biography here ...
        </bio>
    </author>
    <person sex="f">
        <name>Jane Doe</name>
        <bio /> <!-- she has not a biography -->
    </person>
    <person sex="m">
        <name>Richard Roe</name>
        <bio>The antagonist.</bio>
    </person>
</people>
```

The following example illustrates the effect and use of the three operators supposing that the document is stored in a resource file referenced as `sample`.

```haxe
class Main {
    public static function main()
    {
```
Chapter 16: haXe Advanced Topics

```haxe
{ 
    var doc = Xml.parse(Std.resource("sample"));
    var fdoc = new haxe.xml.Fast(doc);
    var people = fdoc.node.people;
    for(n in people.elements)
        trace("person name: " + n.node.name.innerData + " [" + n.name + "]");

    // same collection as before but the node of William Shakespeare
    // is left behind because it is not an element with name person
    for(n in people.nodes.person)
        trace("person name: " + n.node.name.innerData);
    trace("author name: " + people.node.author.node.name.innerData);
}
```

The **innerData** is a handy shortcut to access the textual content (**PCData** or **CData**) of the current element. If the node contains other elements or more than one node, an error is thrown. The **name** variable contains the name of the current element; this value is only available if the node is of the **Element** type (not **Document**).

As querying a node for elements with the wrong name raises an exception, you can check the existence of a certain element using the **hasNode** operator. Remember that the described operators are just plain instance fields.

```haxe
if(people.hasNode.writer)
    trace("writer name: " + people.node.writer.node.name.innerData);
else
    trace("No writers!");
```

### Accessing Attributes

Retrieving attributes is even easier than accessing elements. You can use **has** to check if an attribute exists and **att** to get its value.

```haxe
class Main 
{
    public static function main()
    {
        var doc = Xml.parse(Std.resource("sample"));
        var fdoc = new haxe.xml.Fast(doc);
        var people = fdoc.node.people;
        for(n in people.nodes.person)
        {
            var name = n.node.name.innerData;
            var sex = if(!n.has.sex)
                "unknown"
            else if(n.att.sex == 'm')
                "male"
            else
                "female";
            trace(name + ", " + sex);
        }
    }
}
```

457
When you access nodes and attributes in this way, it is very easy and fast (no pun intended). Nodes and attributes are blended in a seamless way in the code flow.

**Check**

Sometimes a well-formed document is not enough. Before processing a document you must be sure that it is built conforming to some rules. This kind of validation is traditionally performed using DTD or an XML Schema document. The first is quite easy to write and to interpret but it is quite limited in expressiveness; the latter is far more advanced but more complex, too. haXe tries to mediate complexity and richness with a native solution known under the name of Check. The Check class only provides two public static methods: checkNode() and checkDocument(). The first is used to validate the correctness of a generic node and the second is specific for nodes of type Document. The functions do not return anything but throw an error if any validation rule fails; both accept an Xml instance as a first argument and a rule definition as a second. All the magic of the validation is in the defined rules.

*Because the class relies heavily on the EReg class, Check is not available on Flash 6 to 8.*

The following example shows how to check an XML document. The validation rule applied states that the document must have one child with the name root, no attributes, and no children.

```haxe
import haxe.xml.Check;

class Main
{
    public static function main()
    {
        var xml = "<root></root>";
        var doc = Xml.parse(xml);

        var schema = RNode("root");
        try
        {
            Check.checkDocument(doc, schema);
        }
        catch(e : String) {
            trace("Validation failed: " + e);
        }
    }
}
```

Do not forget to import the haxe.xml.Check class because the rules for the validation check are defined in there. In this case only an RNode has been applied and the document is checked using the checkDocument() function. The function is invoked inside a try block because in case of failure an exception is raised.

*All the string comparisons in the Check API are case sensitive; this is particularly important when defining rules for elements and attributes.*

**Checking Nodes**

The nodes are checked using the enum Rule. Rule has several constructors to cover many possible validation conditions.
Chapter 16: haXe Advanced Topics

RNode

The RNode constructor acts on a single node whose element name must match the first argument. The second optional argument is an array of attribute validation rules. If omitted or empty, the node must not have attributes to be valid. The third argument is an optional rule that is applied to the child elements. If omitted, the node must not have child nodes. The constructor has the following signature:

RNode(name:String, ?attribs:Array<Attrib>, ?childs:Rule);

In the following example, the validation rule checks that a node with the element name book has one child element with the name author.

```java
var schema = RNode("book", RNode("author"));
// equivalent to
var schema = RNode("book", [], RNode("author"));
```

Some XML fragments validated using the preceding rule are analyzed in the following code:

```java
// fails because book has no author child node
var xml = "<book></book>";
// validation passes
var xml = "<book><author/></book>";
// fails because book has more than one author child node
var xml = "<book><author/><author/></book>";
```

RMulti

The constructor defines a single validation rule, the first argument that is applied to a sequence of elements. The second argument is a Boolean value that, when true, requires that the sequence must contain at least one node, when false, that the list can be empty. If omitted, the default value is false. The RMulti constructor has the following signature:

RMulti( rule:Rule, ?atLeastOne:Bool );

The following rule requires that an element with the name book must contain an undefined number of subelements with the name author. The sequence can be empty (no elements with the name author) because the second argument of RMulti has been omitted.

```java
var schema = RNode("book", RMulti(RNode("author")));
```

RList

The RList constructor defines many validation rules to apply to a set of nodes. Each rule is tested sequentially if the second argument is true. The default value when the argument is omitted is false. The constructor has the following signature:

RList( rules:Array<Rule>, ?ordered:Bool );
Part III: Extending the Possibilities

The following rule defines a validation rule for a node with the element name `author` that must have subelements `name` and `bio`. If one of the child nodes is missing or the sequence is not correct, the validation fails.

```javascript
var schema = RNode("author", RList([RNode("name"), RNode("bio")], true));
```

**RChoice**

The `RChoice` constructor permits to build complex validation rules. More than one rule can be created and passed in an array as the argument of the constructor and the validation check is successful if at least one of those rules is validated correctly. The constructor has the following signature:

```javascript
RChoice( choices:Array<Rule> );
```

The following rule defines a validation rule for an element that has the name `items` and that contains zero or more elements with the names `movie`, `book`, or `magazine`.

```javascript
var schema = RNode("items", RMulti(RChoice([RNode("movie"), RNode("book"), RNode("magazine")])));
```

The following XML document is valid for the preceding rule:

```xml
<items><book/> <movie/> <book/></items>
```

The following document is not valid because the `toy` element has no matching rule:

```xml
<items><book/> <toy/></items>
```

**ROptional**

The `ROptional` constructor is not a surprise; the rule passed as an argument is applied only if a matching node or set of nodes exists; otherwise it is skipped. The constructor has the following signature:

```javascript
ROptional( rule:Rule );
```

The following validation rule defines that the `items` element must contain two elements, `movie` and `book`, in any order where the `movie` element is optional.

```javascript
var schema = RNode("items", RList([ROptional(RNode("movie")), RNode("book")])));
```

**RData**

The constructor applies a validation rule on the content of nodes of type `CData` or `PCData`. The validation applied is a `Filter` rule that is explained later in this chapter. If the `filter` argument is
omitted, and the validator only requires that the node must be of the correct type. The `RData` constructor has the following signature:

```javascript
RData(?filter:Filter);
```

The following rule states that the element with the name `div` must contain some textual data:

```javascript
var schema = RNode("div", RData());
```

The following XML fragment is valid:

```xml
<div> test </div>
```

The following two pieces of code are not valid because the first is empty and the second contains an extra element node:

```xml
// empty element
<div/>

// PCData + Element content
<div> test <span></span></div>
```

### Checking Attributes

The `Attribute` enum has only one constructor `Att` that, as the name suggests, is used to validate a node attribute. The only `Rule` that accepts an attribute constraint is the `RNode` constructor. The sequence of the attribute rules is not relevant and all the attribute rules must match for the node to be valid. The constructor accepts a first mandatory argument that is the name of the attribute that has to be checked. The second argument is an optional `Filter` value that acts on the value of the attribute. The third argument is a default value for the attribute when this is missing. That is, if the attribute rule has a default value, the attribute becomes optional and when it does not exist in the node, it is automatically created; if the default value is omitted the attribute is mandatory. The `Filter` condition is checked only if the attribute exists at the moment of the validation. The `Att` constructor has the following signature:

```javascript
Att(name:String, ?filter:Filter, ?defvalue:String);
```

The following rule defines that the element to validate must have a name `person` and an attribute with name equal to `name`:

```javascript
var schema = RNode("person", [Att("name")]);
```

The following XML fragment validates correctly, but so does the one with an empty attribute. The rule just states that the attribute must exist and makes no assumptions about its content. To apply a validation on the attribute value, see the next section about filtering values.

```xml
// valid
<person name="John Doe" />

// valid, too
<person name="" />
```
Part III: Extending the Possibilities

When the attribute rule also specifies a default value, the content of the original node can be modified during validation. For the following rule the node must have an attribute with the name `sex`; if the attribute is not present, a default one is automatically created with value `m`.

```javascript
var schema = RNode("person", [Att("name"), Att("sex", "m")]);
```

Therefore, the first node `person` from the previous example is valid for the above rule but is changed as follows:

```xml
<person name="John Doe" sex="m"/>
```

**Filtering Values**

Filter rules can be applied to nodes of type CData or PCData or to attribute values. The `Filter` enum has four very intuitive constructors. The first `FInt` checks for string values that can be converted to integer values. The `FBool` checks that the passed string can be converted to a Boolean value; accepted values are true, false, 0, and 1 and they are case sensitive. The `FEnum` constructor accepts an array of string values; if one in the list matches the tested value, the validation is passed. Finally, the `FReg` constructor accepts a regular expression, an `EReg` object, to validate the string value.

The following rule adds two filters to the previous attribute example. The first `FReg` rule checks that the name attribute is composed only of words that start with an uppercase character, separated by a single space and it excludes any character not in the alphabet. The `sex` attribute is also validated against the only two possible values `f` and `m` using an `FEnum` rule.

```javascript
var schema = RNode("person", [Att("name", FReg(~/^[A-Z][a-z]*( [A-Z][a-z]*)*$/)),
Att("sex", FEnum(['f', '"m"'], "m"))]);
```

See how the following XML fragments behave when validated against the preceding rule:

```xml
// passes
<person name="John Doe" sex="m"/>
// passes, too
<person name="John Doe"/>
// fails because the "." is not allowed in attribute name
<person name="John D." sex="m"/>
// fails because "x" is not in the list of the acceptable values for sex
<person name="John Doe" sex="x"/>
```

As already mentioned, the filters can also be applied to Character Data contents. In the following example, the rule applies to an element node with the name `person` that has an attribute `name`, and has two subnodes `age` and `male` that must contain respectively a textual node that represents an integer number and a value that can be converted to a Boolean value.

```javascript
var schema = RNode("person",
[Att("name", FReg(~/^[^\d.]{3,20}$/)), // any string of 3 to 20 characters
RList([ RNode("age", RData(FInt)),
RNode("male", RData(FBool)),
], true));
```
See how the following XML fragments behave when validated against the preceding rule:

- // passes the validation
  <person name="John Doe"> <age>35</age> <male>true</male></person>
- // passes, too
  <person name="John Doe"> <age>-1</age> <male>1</male></person>
- // fails because age is not numeric
  <person name="John Doe"> <age>x</age> <male>true</male></person>
- // fails because male has wrong casing
  <person name="John Doe"> <age>35</age> <male>True</male></person>

Check is certainly a useful tool that is probably still underestimated in the haXe community. It is easy to create complex rules, by far easier than writing an equivalent in XML Schema and for sure more complex rules than the ones definable with DTDs. The use of regular expressions in the rules is really fundamental; more information on the regular expressions can be found in Chapter 8 in the “Regular Expressions” section.

**XML Proxy**

In version 1.14 of haXe, the `haxe.xml.Proxy` class has been added. The idea behind the class is to add a type check at compile time that is based on an XML resource. The proxy class must always be extended to be used, and when you are extending a `haxe.xml.Proxy`, two type parameters must be declared. The first is a `Const` type parameter whose value must be the filename of an existing XML file. The XML file must be reachable at compile time and, thus, can be placed anywhere as far as the folder is in the compiler class path. The second type parameter denotes the type of the fields defined by the proxy.

In the following example, an XML Proxy is used to be sure that the data loaded from the XML file is really available and can be used in code. The XML file `countries.xml` has the following data and structure:

```
<countries>
  <country id="italy"
    country="Italy"
    lang="it"
    currency="euro" />
  <country id="canada"
    country="Canada"
    lang="en,fr"
    currency="canadian dollar" />
  <country id="usa"
    country="United States of America"
    lang="en"
    currency="dollar" />
</countries>
```

In the haXe code, the referenced file is used two times, once to set the compiler constraint in the XML Proxy and the other to load the file as a resource; don’t forget to add the `-resource countries.xml` switch in your compiler command.
Part III: Extending the Possibilities

class LocaleProxy extends haxe.xml.Proxy<"countries.xml", Locale> {
    class Main {
        public static function main()
        {
            var locales = getLocales();
            trace(locales.italy.country);
            //trace(locales.uusa.country); // does not compile because misspelled
        }
        public static function getLocales()
        {
            var doc = Xml.parse(Std.resource("countries"));
            var h = new Hash();
            for(n in doc.firstElement().elements())
            {
                h.set(n.get("id"),
                {
                    country : n.get("country"),
                    languages : n.get("lang").split(",
                    currency : n.get("currency")
                });
            }
            var p = new LocaleProxy(h.get);
            return p;
        }
    }

typedef Locale = {
    country : String,
    languages : Array<String>,
    currency: String
}

This is the third type of proxy class that is described in the book. The first and the second, the haxe .remoting.Proxy and haxe.remoting.AsyncProxy from the Remoting API, are discussed in Chapter 15.

Reflection API

Reflection in computer science is the capability of changing or inspecting the type definitions at run time (during the execution of the program). Just to cite some uses for reflection, it is possible to set a value for an object field just knowing its name at run time, or to create an instance without using the new keyword and also without invoking its constructor.

In the reflection area haXe provides some useful classes; the two most important ones are Reflect and Type.

The Reflect Class

The Reflect class from the standard library is very handy to handle object instances at a very low level and it contains the static methods detailed in the Table 16-1.
<table>
<thead>
<tr>
<th>Static Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>callMethod(o:Dynamic, func:Dynamic, args:Array&lt;Dynamic&gt;) : Dynamic</td>
<td>Invokes a method with the passed object as scope and an array of values as arguments.</td>
</tr>
<tr>
<td>compareMethods(f1:Dynamic, f2:Dynamic) : Bool</td>
<td>Returns true if f1 and f2 are references to the same static method or to the same method of the same instance.</td>
</tr>
<tr>
<td>copy&lt;T&gt;(o:T) : T</td>
<td>Creates a clone of the passed object. The function is not recursive.</td>
</tr>
<tr>
<td>deleteField(o:Dynamic, f:String) : Bool</td>
<td>Removes the field from the object. This is not the same as assigning null to the field.</td>
</tr>
<tr>
<td>empty() : ( )</td>
<td>Creates an empty object. This object doesn’t instantiate any concrete class.</td>
</tr>
<tr>
<td>field(o : Dynamic, field:String) : Dynamic</td>
<td>Returns the field of the object or null if it does not exist. If the field is a function, remember that once extracted the original instance scope may be lost.</td>
</tr>
<tr>
<td>fields(o:Dynamic) : Array&lt;String&gt;</td>
<td>Returns an array of strings containing the names of the fields of the passed object. The array doesn’t contain the class methods and the uninitialized variables.</td>
</tr>
<tr>
<td>hasField(o:Dynamic, field:String) : Bool</td>
<td>Returns true if the passed object contains a field named as the second argument; returns false otherwise. The method doesn’t take into account class methods and uninitialized variables.</td>
</tr>
<tr>
<td>isFunction(f:Dynamic) : Bool</td>
<td>Returns true if the argument is a function; returns false otherwise.</td>
</tr>
<tr>
<td>isObject(v:Dynamic) : Bool</td>
<td>Returns true if the argument is an object; returns false otherwise.</td>
</tr>
<tr>
<td>makeVarArgs(f:Array&lt;Dynamic&gt; -&gt; Dynamic) : Dynamic</td>
<td>Transforms a function that takes an array of arguments into a function that can be called with any number of arguments.</td>
</tr>
<tr>
<td>setField(o:Dynamic, field:String, value: Dynamic) : Void</td>
<td>Sets the value of the field with the specified name on the passed object.</td>
</tr>
</tbody>
</table>

Table 16-1
Part III: Extending the Possibilities

To invoke a method knowing its name at run time, you can use the following code snippet:

```javascript
var a = Reflect.callMethod(ob, Reflect.field(ob, "execute"), ["abc", 123]);
```

Where `ob` is an object that has a method `execute()` that accepts two arguments, a string, and an integer value.

When you use the `field()` method, it is possible to obtain a variable value or a reference to a function.

```javascript
var f = Reflect.field(ob, "execute");
var a = f(); // the object scope is preserved
var n = Reflect.field(ob, "name"); // "name" is a variable
Reflect.deleteField(ob, "name");
trace(Reflect.hasField(ob, "name")); // trace "false"

// this may not work as expected on class instances in Flash 9
Reflect.setField(ob, "name", "John");
trace(Reflect.hasField(ob, "name")); // trace "true"
```

The `makeVarArgs()` method is quite interesting. It transforms a function as shown in the following example:

```javascript
var f = function(args : Array<Dynamic>)
{
    return args[0] + args[1] + args[2];
}
var fun = Reflect.makeVarArgs(f);
trace(fun(1, 2, 3));
```

Although the `Reflect` class is a very powerful tool, it must be used with caution in many situations. Imagine for example that a variable is defined in this way:

```javascript
class Person
{
    public var name(getName, setName) : String;
    //...
}
```

Outside the `Person` class, the field is accessed using the identifier `name`; but what happens if the following code is executed?

```javascript
trace(Reflect.field(o, "name")); // where o is an instance of Person.
```

It will always trace to `null` even when `getName()` returns a not-empty value. This is because in this class the `name` variable is just a syntax shortcut and it is not compiled in the resulting file.

The same happens in this situation:

```javascript
var employee : Dynamic = new Person();
trace(employee.name);
```

If reflection is used in a cross-platform environment, it is important to test these functionalities on each targeted platform individually because the reflection API makes use of low-level platform-specific calls and some inconsistencies can possibly occur.
Chapter 16: haXe Advanced Topics

The Type Class

The Type class works on class definitions as the Reflect class works on object instances. Its methods, which are also all static, are described in Table 16-2.

Table 16-2

<table>
<thead>
<tr>
<th>Static Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createInstance&lt;T&gt;(cl:Class&lt;T&gt;, args:Array&lt;Dynamic&gt;) : T</td>
<td>Instantiates the class T using the passed arrays as constructor arguments.</td>
</tr>
<tr>
<td>createEmptyInstance&lt;T&gt;(cl:Class&lt;T&gt;) : T</td>
<td>Instantiates the class T without calling the class constructor.</td>
</tr>
<tr>
<td>enumConstructor(e:Dynamic) : String</td>
<td>Returns the constructor name of the passed Enum.</td>
</tr>
<tr>
<td>enumEq&lt;T&gt;(a:T, b:T) : Bool</td>
<td>Compares recursively the constructors of two Enum of type T and their parameters.</td>
</tr>
<tr>
<td>enumIndex(e : Dynamic) : Int</td>
<td>Returns the index of the constructor of an enum.</td>
</tr>
<tr>
<td>enumParameters(e:Dynamic) : Array&lt;Dynamic&gt;</td>
<td>Extracts the constructor parameters of the passed Enum and returns them in a Dynamic array.</td>
</tr>
<tr>
<td>getClass&lt;T&gt;(o:T) : Class&lt;T&gt;</td>
<td>Returns the class definition for the class of type T.</td>
</tr>
<tr>
<td>getClassFields(c:Class&lt;Dynamic&gt;) : Array&lt;String&gt;</td>
<td>Returns an array of strings containing the names of all the static fields defined in the passed class definition.</td>
</tr>
<tr>
<td>getClassName(c:Class&lt;Dynamic&gt;) : String</td>
<td>Returns the complete name of the passed class definition.</td>
</tr>
<tr>
<td>getInstanceFields(c:Class&lt;Dynamic&gt;) : Array&lt;String&gt;</td>
<td>Returns an array of strings containing the names of all the instance fields defined in the passed class definition.</td>
</tr>
<tr>
<td>getEnum(o:Dynamic) : Enum</td>
<td>Returns the enum definition of the passed value if this is an Enum instance; returns null otherwise.</td>
</tr>
<tr>
<td>getEnumConstructs(e:Enum) : Array&lt;String&gt;</td>
<td>Returns an array of strings containing all the names of the constructors available in the argument Enum. Starting from version 1.16 the sequence of the constructors is guaranteed.</td>
</tr>
<tr>
<td>getEnumName(e:Enum) : String</td>
<td>Returns the complete name of the passed Enum definition.</td>
</tr>
</tbody>
</table>

Table continued on following page
Part III: Extending the Possibilities

<table>
<thead>
<tr>
<th>Static Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSuperClass(c:Class&lt;Dynamic&gt;) : Class&lt;Dynamic&gt;</td>
<td>Returns the super class definition of the passed class definition.</td>
</tr>
<tr>
<td>resolveClass(name :String) : Class&lt;Dynamic&gt;</td>
<td>Evaluates the passed string argument and returns a matching class definition or null if it doesn’t exist. The Enum must have been compiled in the file to be accessible.</td>
</tr>
<tr>
<td>resolveEnum(name:String):Enum</td>
<td>Evaluates the passed string argument and returns a matching Enum definition or null if it doesn’t exist. The Enum must have been compiled to be accessible.</td>
</tr>
<tr>
<td>toClass(t:Dynamic) : Class&lt;Dynamic&gt;</td>
<td>Returns a class definition if the passed value is a class; returns null otherwise.</td>
</tr>
<tr>
<td>toEnum(t:Dynamic) : Enum</td>
<td>Returns an Enum definition if the passed value is an Enum; returns null otherwise.</td>
</tr>
<tr>
<td>typeof(v:Dynamic) : ValueType</td>
<td>Returns the type of the passed argument.</td>
</tr>
</tbody>
</table>

**ValueType** is an **Enum** with the following constructors:

```
TNull;
TInt;
TFloat;
TBool;
TObject;
TFunction;
TClass( c : Class<Dynamic> );
TEnum( e : Enum );
TUnknown;
```

With **Type**, an object can be instantiated using just the class name. This can be done in two ways using the **createInstance()** and the **createEmptyInstance()**.

```javascript
class Vehicle
{
    public static function main()
    {
        var cl = Type.resolveClass("Vehicle");
        var v1 : Vehicle = Type.createInstance(cl, ["motorcycle"]);
        trace(v1.type); // trace "motorcycle"
        var v2 : Vehicle = Type.createInstance(cl, [null]);
        trace(v2.type); // trace "car"
        var v3 : Vehicle = Type.createEmptyInstance(cl);
        trace(v3.type); // trace null
    }
```

468
The `resolveClass()` returns a `Class` type that is used to identify the class. You have to specify the full path with correct dot syntax if the class you want to resolve is contained in a subpackage. Note the differences between `createInstance()` and `createEmptyInstance()`: The first creates an object using the `new` statement and therefore also needs the constructor arguments while the other builds the instance without invoking the constructor. This can be handy when an object has been serialized or stored to some media and must be rebuilt without invoking a second initialization.

Another common operation performed by the `Type` class is to discover the types of the objects.

```hx
class TypeSample
{
    public static function main()
    {
        // an array of items of different types
        var objs = [
            RGB(255, 200, 0),
            new Person("John"),
            { text : "I am anonymous" },
            White,
            10
        ];
        for(ob in objs)
        {
            switch(Type.typeof(ob))
            {
                case TClass(cl):
                    trace("Instance of " + Type.getClassName(cl));
                case TEnum(e):
                    trace("Enum " +
                        Type.getEnumName(e) + "." +
                        Type.enumConstructor(ob) +
                        "(" + Std.string(Type.enumParameters(ob)) + ")");
                case TObject:
                    trace("Anonymous object");
                default:
                    trace("Unsupported type");
            }
        }
    }
}
```

// traces (PosInfos information has been removed):
// Enum Color.RGB([255,200,0])
// Instance of Person
// Anonymous object
// Enum Color.White([])
// Unsupported type
Runtime Type Information (RTTI)

The Reflect and Type classes can be formidable to manage instances and methods at run time using name references, but there is a lot of information that you cannot obtain using those classes; how do you obtain the arguments of a function and their types? How can you guess the structure of an object defined using a typedef? The answer to those questions is just an acronym: RTTI (Runtime Type Information).

The RTTI information is stored in one or more XML documents that can be parsed to use in code. The XML document is the same one used by the haxedoc tool described in Chapter 6 to generate the code documentations. Accessing this file at run time gives you a complete overview of all the classes and types included in the compiled project. The easiest way to access this information is passing the XML document to the `process()` function of the `haxe.rtti.XmlParser` class. The method transforms the XML file in an array of definitions; each definition describes one type, including packages, using an enum constructor. The enum for those definitions is `TypeTree` (Table 16.3) that is defined in the `haxe.rtti.Type` module.

Table 16-3

<table>
<thead>
<tr>
<th>Enum TypeTree Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPackage(name:String, full:String, subs:Array&lt;TypeTree&gt;)</td>
<td>This constructor is used for a package. It contains its name, the full path, and an array of the types defined in the package.</td>
</tr>
<tr>
<td>TClassdecl(c:Class)</td>
<td>This type is a class.</td>
</tr>
<tr>
<td>TEnumdecl(e:Enum)</td>
<td>This type is an enum.</td>
</tr>
<tr>
<td>TTypedecl(t:Typedef)</td>
<td>This type is a typedef.</td>
</tr>
</tbody>
</table>

When looking at the `haxe.rtti` API, please note that some of the type names there defined conflict with names defined in the default package. If you have to use both of them in the same code context, you will probably want to avoid conflicts using typedef to create proper aliases.

The following simple function processes an XML RTTI Document:

```haxe
class { public static function parse(file) { var xml = neko.io.File.getContent(file); var doc = Xml.parse(xml); var parser = new haxe.rtti.XmlParser(); parser.process(doc.firstElement(), "neko"); // set the platform you need return parser.root; } }
```

The `parser.process()` function accepts an XML node as an argument and a platform name. The processed structure is contained in the root field of the `XmlParser` instance and is an array of `TypeTrees`. Each package is a container for subtypes and types can be `Class`, `Enum`, and `Typedef` enumerations.
Another way to access RTTI information without loading an external XML file and available only for class instances, is to use the `haxe.rtti.Infos` interface. A class that extends it is generated with a new static field with name `__rtti` that contains the full XML notation for the current class.

The following example shows a class that implements `Infos`:

```haxe
/**
 * Test Class.
 */
class Inspected implements haxe.rtti.Infos
{
    public static function main()
    {
        var i = new Inspected();
        trace(i.inspect());
    }
    public var name(default, setName) : String;
    public function new() { }
    private function setName(n : String)
    {
        return name = n;
    }
    public function inspect()
    {
        var p = new haxe.rtti.XmlParser();
        // the access to __rtti must be untyped because the field does not exist
        // at compile time.
        return p.processElement(Xml.parse(untyped Inspected.__rtti).firstElement());
    }
}
```

The `inspect()` method processes the content of the XML document contained in `Inspected.__rtti` and returns it as an enumerator that describes the class. In the following code, the output of the `trace()` invocation has been formatted and commented to show how the information is returned.

```
TClassdecl{
    // an object that contains the Class-specific information
    {
        // the file name without extension where the type is declared
        // the module is null if the type name corresponds to the filename
        module => Analyzer,
        // the full path including package names separated by a dot '.'
        path => Inspected,
        // a List of implemented interfaces
        interfaces => [{ path => haxe.rtti.Infos, params => [] }],
        // class type parameters in an array of string
        params => [],
        // the documentation for the current type
        // this is not parsed and included when the Infos interface is used
        doc => null,
        // a List of fields that belong to the type
        fields => {
            (continued)
```
Part III: Extending the Possibilities

(continued)

{ // this field is the variable "name"
  // the field name
  name => name,
  // the field type
  type => TClass(String,{}),
  // the field type parameters
  params => null,
  doc => null,
  // get accessor (default is RNormal that means no accessor)
  get => RNormal,
  // set accessor for a setter method
  set => RMethod(setName),
  platforms => {},
  // public or private?
  isPublic => true },

{ // this field is the private method "setName"
  name => setName,
  type => TFunction(
    // a list of arguments
    {{
      // the function accepts one argument with name n ...
      name => n,
      // ... of type string ...
      t => TClass(String,{}),
      // ... that is not optional
      opt => false }),
    // the return type of the function (string)
    TClass(String,{})),
  params => null,
  doc => null,
  get => RNormal,
  set => RNormal,
  platforms => {},
  isPublic => false },

{ // the instance method inspect()
  name => inspect,
  type => TFunction(
    // the method accepts no arguments
    {},
    // and returns an enumeration ...
    TEnum( haxe.rtti.TypeTree,
      // ... with no type parameters
      {})),
  params => null,
  doc => null,
  get => RNormal,
  set => RNormal,
  platforms => {},
  isPublic => true ),
{  
    // the class constructor  
    // it is a normal function after all  
    name => new,  
    type => TFunction({},TEnum(Void,{})),  
    params => null,  
    doc => null,  
    get => RNormal,  
    set => RNormal,  
    platforms => {},  
    isPublic => true },  
  
  // a type can be private for its module  
  isPrivate => false,  
  platforms => {},  
  isInterface => false,  
  // has the class the extern modifier?  
  isExtern => false,  
  // this class has no ancestors ...  
  superClass => null,  
  // ... and does not implement Dynamic<T>  
  dynamic => null,  
  statics => {  
    // a list of static fields  
    { name => main,  
      type => TFunction({},TEnum(Void,{})),  
      params => null,  
      doc => null,  
      get => RNormal,  
      set => RNormal,  
      platforms => {},  
      isPublic => true }  
  })

Serialization

Serialization is the process of transforming an object into a binary sequence and deserialization, or unserialization, is the process of restoring the original object from the binary state. Serialization is mainly used in two areas: transmission of information and data storage. The advantage of serialization, and particularly the one provided by the haXe API, is that entire object graphs can be serialized in one step.

The haXe Serialization API uses a proprietary string format to transform objects into a medium storable sequence of characters. Almost any kind of haXe type can be serialized and deserialized; the most notable exceptions are platform-native objects (you cannot really serialize a Flash movie clip or a Neko database connection object) and functions. Another area where serialization does not play well is events management because the serialization process can potentially add reference objects that must not go in the serialized object graph and the unserializer could not be able to re-wire the events properly during the object reconstitution. Apart from those cases the serialization process is fast and efficient particularly in the Neko platform where the code is optimized using a native implementation.
Part III: Extending the Possibilities

The Serializer class has a static variable USE_CACHE whose default value is set to false that determines if the serialization process must account on recursive object references. It is fundamental to set this variable to true if you need to serialize an object graph that has parent/child bidirectional relations; otherwise the serialization process can end in an infinite loop.

As anticipated before, the haxe.Serializer and haxe.Unserializer classes can handle all the types available in haXe including enum, class instances, and anonymous objects. The Serializer class has one serialize() method that accepts an argument of type Dynamic that is the object or value to serialize. Once an object has been serialized, the serialized string value can be obtained using the toString() instance method. A shortcut to the whole process can be performed using the static run() method that accepts a Dynamic value as an argument and returns it in the serialized form.

The Unserializer class does the contrary; an object in a serialized string form is passed as the constructor argument and the unserialize() method rebuilds the object to its original form. The unserialize() function is not aware of the type of object that it is going to process and returns a generic Dynamic object; to use it properly you have to cast it to its original type. This class also has the shortcut run() method that accepts an object in the serialized form as an argument and returns it expanded to its original form.

Please note that the code that is delegated to unserialize the object must contain the definition of the object type or an error is thrown. The unserialization process is also quite permissive about type definition changes, but be aware that renaming a field in a class definition after the serialization can break the process. In general, adding fields and changing functions do not create problems.

Another important matter to note is that the unserialize() method does not invoke the class constructor for object instances. This can have some effects on field definitions introduced after the first serialization or other initialization code.

The following snippet shows how to serialize/deserialize an anonymous object:

```haxe
class Main
{
    public static function main()
    {
        var ob = {
            name : "John Doe",
            age : 35
        };
        var ser = haxe.Serializer.run(ob);
        // trace something like:
        // oy4:namey10:John%20Doey3:agei35g
        trace(ser);
        var desob : Person = haxe.Unserializer.run(ser);
        trace(desob.name + ": " + desob.age);
    }
}
typedef Person = {
    name : String,
    age : Int
}
```

```haxe```
haXe Magic

Some functionalities in haXe are platform specific, others violate or at the very least circumvent the canonical syntax definitions. Those features are called magic. They are structures or keywords that break the standard boundaries of the haXe syntax to give some advantages to the developer.

Not all the magic features are described in this section, because they are also subject to changes, being low-level constructs; another reason is that many of these magic features are already wrapped in a more convenient and object-oriented definition. Here is a description of the constructs you will most likely need to use and the ones that are unlikely to be changed in future versions. You can find more information on haXe magic on the official website at www.haxe.org/.

Platform Independent

Some of the haXe magic is platform independent and is described next; other magic exists for classes/interfaces and inheritance hierarchies, but it is very unlikely that you need to deal with it directly.

Boot Classes

Every time a project is compiled, an additional Boot class is included therein; the class contains code that is executed in the bootstrap phase and that is used to unify the platform-specific differences. The developer will not usually use this class directly and may not even be aware of its existence because the haXe environment treats it transparently.

Static Initialization

Every class can have a static method __init__() that is executed automatically before the main() method invocation. Inside this method you cannot use the try/catch construct because the needed declarations required to make the construct work may not be initialized yet. Another thing that requires caution is to avoid using values from other static fields because their initialization is not guaranteed to occur before the __init__ execution.

__resolve and __setfield

Every class that implements Dynamic automatically gains two magic methods: __resolve() and __setfield(). The __resolve() method is auto-magically invoked each time that an inexistente instance field is used. If the field is a method, it is probably a good practice to use __resolve to return a function instead of a value. The returned function is then used to perform the proper action. When a matching field exists in the instance it behaves normally.

```plaintext
class Resolver implements Dynamic
{
    public static function main()
    {
        var r = new Resolver();
        r.inexistant();
        r.exist();
    }
}
```

(continued)
Part III: Extending the Possibilities

(continued)

```javascript
public function new() { }

function __resolve(method : String) : Dynamic
{
    return function()
    {
        trace("Invoked the method: " + method);
    }
}

public function exist()
{
    trace("I exist");
}
```

To handle the method arguments, you can use the `Reflect.makeVarArgs()` method:

```javascript
class Resolver implements Dynamic
{
    public static function main()
    {
        var r = new Resolver();
        r.inexistant("Hello", "world");
    }
    public function new() { }
    function __resolve(method : String) : Dynamic
    {
        return Reflect.makeVarArgs(function(args) {
            trace("Invoked the method: " + method);
            trace("with arguments: " + args[0] + " " + args[1]);
        });
    }
}
```

The `__setfield()` magic method works similarly but it is executed when the value of an instance variable is set and the variable is not defined in the class. The following example uses `__setfield()` and `__resolve()` to create a dynamic getter and setter for every field that is not defined in the class:

```javascript
class MixedHash implements Dynamic
{
    public static function main()
    {
        var r = new MixedHash();
        r.name = "John"; // __setfield not executed
        r.age = 35; // __setfield executed
        trace(r.name);
        trace(r.age);
        // trace(r.inexistant); // throws an error
    }
    public function new()
    {
        h = new Hash();
    }
}
```

476
Chapter 16: haXe Advanced Topics

```haxe
public var name : String;
private var h : Hash<Dynamic>;
function __setfield(name : String, value : Dynamic)
{
    trace(name + " " + Std.string(value));
    h.set(name, value);
}
function __resolve(field : String)
{
    return if(h.exists(field))
        h.get(field)
    else
        throw "unexistent field " + field;
}
}
```

**Infos Interface**

The `haxe.Infos` interface, already explained in the RTTI section of this chapter, adds an `__rtti` static variable to the class declaration that contains the XML description of the class itself.

**Public Interface**

The fields of a class that implement the `haxe.Public` interface are all public by default if the `public/private` modifier is omitted. This is the opposite of the standard behavior where an access modifier is `private` by default.

**PosInfo**

Every time a function that has an optional last argument of type `PosInfo` is used without passing a value to that argument, this is automatically filled with an object that contains the current filename, the line number, and the class and method in use. The `haxe.PosInfo typedef` is fundamental for debugging purposes and it is used in the `trace()` function and other code related functions as the unit testing ones.

**Neko**

Neko has a collection of operations that are optimized by the compiler known as `builtins`. Each `builtin` in the Neko language is prefixed with the dollar symbol (`$`). An example of `builtin` is `$scopy()` that takes a string as an argument and returns a copy of the same as the result. For the full list of the available `builtins`, look at the dedicated page on the Neko website at [http://nekovm.org/doc/view/builtins](http://nekovm.org/doc/view/builtins).

In haXe you can use the very same operations, but the dollar prefix must be replaced with `__dollar__`. So the aforementioned `$scopy()` is available as `__dollar__scopy()`.

**Flash**

A set of Flash-specific keywords are defined and enclosed between double underscore symbols, as shown in Table 16-4. The names are directly mapped to the Flash API and need no description.
Part III: Extending the Possibilities

Table 16-4

<table>
<thead>
<tr>
<th>Magic Statement</th>
<th>Flash Version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>arguments</strong></td>
<td>Flash 6 to 9</td>
</tr>
<tr>
<td><strong>as</strong>(e, t)</td>
<td>Flash 9</td>
</tr>
<tr>
<td><strong>delete</strong>(o, f)</td>
<td>Flash 6 to 9</td>
</tr>
<tr>
<td><strong>eval</strong>(e)</td>
<td>Flash 6 to 8</td>
</tr>
<tr>
<td><strong>gettimer</strong>()</td>
<td>Flash 6 to 8</td>
</tr>
<tr>
<td><strong>geturl</strong>(e1, e2)</td>
<td>Flash 6 to 8</td>
</tr>
<tr>
<td><strong>global</strong></td>
<td>Flash 9</td>
</tr>
<tr>
<td><strong>instanceof</strong>(e1, e2)</td>
<td>Flash 6 to 8</td>
</tr>
<tr>
<td><strong>in</strong>(e, f)</td>
<td>Flash 9</td>
</tr>
<tr>
<td><strong>is</strong>(e, t)</td>
<td>Flash 9</td>
</tr>
<tr>
<td><strong>keys</strong>(e)</td>
<td>Flash 9</td>
</tr>
<tr>
<td><strong>new</strong></td>
<td>Flash 6 to 9</td>
</tr>
<tr>
<td><strong>random</strong>(e)</td>
<td>Flash 6 to 8</td>
</tr>
<tr>
<td><strong>trace</strong>(e)</td>
<td>Flash 6 to 8</td>
</tr>
</tbody>
</table>

For further information on the Flash platform, see Chapter 12.

JavaScript

The JavaScript environment has just one magic function __js__. Whatever is passed to this function is treated as native JavaScript code and passed as is in the output file. The argument of this special function cannot be dynamic and must be a hard coded string literal.

Summary

The chapter covered a lot of advanced topics that render the haXe environment unique, efficient, and easy to afford. The topics covered were as follows:

- How to deal in a more efficient way with XML files
- How the Reflection API is structured
- What RTTI is and how to use it in haXe
- How to Serialize/Unserialize objects
- Platform independent and dependent haXe Magic

In the next chapter, you learn how to build desktop applications using Neko. Neko can be used not only to build server-side applications, but also command-line tools and applications with sophisticated GUIs.
Desktop Applications with Neko

As you’ve seen previously, the Neko Virtual Machine provides great flexibility for the development of applications for execution across the Internet, or with the use of a command window. However, the level of flexibility does not stop there. With the help of external libraries and tools, the Neko virtual machine can be used to produce highly portable and incredibly powerful desktop applications that can compete against any developed with a higher level language.

In this chapter, you’ll look closely at how desktop applications can be developed in haXe for Neko, with particular focus on the following:

- What options are available for desktop applications development with Neko
- How to create an executable from a Neko .n file
- What GUI frameworks are available to the Neko developer
- How to use the nGUI portable GUI extension
- An introduction to using the hxGtk portable GUI extension
- How to use Glade with hxGtk

Why Use Neko for Desktop Applications?

Numerous languages are available that provide support for the development of desktop applications. Many of these languages provide low-level integration into the Operating Systems API, or provide a framework utilizing best practices or cross runtime communications. The problem often arises, however, that such languages can be a little cumbersome and leave little to be desired, especially when developing small nonenterprise applications.

Neko’s flexibility, extendibility, and incredibly small footprint make it an ideal candidate for replacing many larger language frameworks. Its ability to perform almost any task with the help of...
Part III: Extending the Possibilities

extensions, alongside its incredible stability, means that you might never find a limitation for its use. If a limitation in Neko’s API does become apparent, then you can simply extend it to fill that limitation.

The current trends in today’s software development world show that browser-based Internet applications are slowly consuming their static desktop equivalents thanks to an increase in technology and inspired development strategies. Now, this is not to say that all applications are becoming browser based, but certainly where business logic is usually concerned, the Web offers a great deal of advantages to company communications that are too good to resist. This inevitable change in software development trends, however, does not hide the reality that desktop run, nonbrowser-based applications are still a necessary essential, of which the Neko Virtual Machine, despite being designed to facilitate Internet-based applications, has little trouble accommodating.

Creating a Neko Executable

Okay, so you’ve developed a great Neko script that performs business logic on your client’s desktop. You’ve left instructions on how the application should be used, and you’ve provided the necessary training for the client. The only problem is, having your client enter `> neko someScript` at a command line doesn’t exactly look very professional. Thankfully, the standard Neko installation comes with an executable tool called NekoTools. You may remember this tool from a previous chapter, when it was used to help construct a web server for debugging your Neko scripts. This tool also provides another handy feature — the ability to convert Neko scripts, ending in `.n`, to an executable file.

Performing this feat is incredibly simple. Navigate to the `.n` file and, at the command prompt, enter:

```
nekotools boot < filename.n>
```

This creates an executable file of the same name in the same directory. It does this by appending the bytecode stored in the `.n` file to the end of the `neko.exe` application used to execute your scripts and saving it as a new file. `neko.exe` has been constructed to first check for code appended to itself before executing an external script.

You can now use the application as you would any other executable, though you must remember to include the Neko runtime files, found in the Neko installation directory, when distributing your application.

Graphical User Interface Libraries

You now know how to create an executable shell application with Neko, but this might not be how you’d like your application to be interacted with. As you may know, many scripting languages and virtual machines similar to Neko, such as Ruby or Python, provide graphical user interface extensions, or GUI extensions. Neko is no different and currently has several options available to meet this end.

**ScreenWeaver HX**

The first and foremost of these extensions is the ScreenWeaver HX library. This extension uses Flash as the GUI interface layer while using the haXe Remoting framework for communications, thus providing infinite possibilities for graphical interfaces. ScreenWeaver HX is unique and deserves your undivided attention, so you look at this library in the next chapter.
Chapter 17: Desktop Applications with Neko

**Xinf: Xinf Is Not Flash**

Xinf, a recursive acronym for Xinf Is Not Flash, is a cross-platform, cross-runtime framework for GUI development. Utilizing OpenGL for rendering controls with the Neko platform, Xinf promises a means to develop applications that look and work in the same manner, whether deployed for Flash, JavaScript, or Neko. In essence, it is a *develop once, deploy anywhere* framework. Unfortunately, Xinf only has one developer, its creator Daniel Fischer, who, while dedicated, has little manpower to really get the library off the ground anytime soon. Thus, at the time of writing this book, there is little in the way of functionality that could be perceived as more than an ambitious, if impressive, project.

**nGui**

The next available GUI framework for Neko is the nGui library. This is our own bindings to the Tecgraf IUP open source library and is usable with the Apple Mac OSX, Linux Motif and GTK, and the Windows operating systems. nGui provides a limited set of functionality, but benefits from its use of native controls within each operating system, providing powerful functionality while requiring minimal resources both in terms of module file size and runtime memory. The haXe class framework for this library has been developed to be simple and intuitive to use, so it presents a great prototyping platform.

At the time of writing this book, this library provides binary ndlls for Windows only, though the project is also compilable for Linux and Mac OSX, should you know how.

**hxGtk**

Finally, you have the hxGtk library developed by Ritchie Turner (aka Blackdog) of iPowerHouse.com. This is probably the most complete of any of the available GUI libraries for Neko, with the exception of ScreenWeaver HX. hxGtk provides bindings to the GTK framework, normally associated with Linux, but is available on all operating systems supported by Neko. Unfortunately, GTK does have a file size issue on the Windows platform, as the dynamic libraries required for the GTK distribution to work can raise an application’s file size by nearly 15MB. However, size isn’t everything, and if this drawback doesn’t pose an issue, then hxGtk is definitely the way to go for now, providing all manner of controls and catering for most application GUI requirements.

**Installing the Required Library**

Each of the GUI libraries mentioned in this chapter can be installed using the haxelib tool. Simply open a command window or terminal and type

```
> haxelib install nGui
```

or

```
> haxelib install hxGtk
```

This installs the library to the *lib* directory of your haXe installation. Then, to compile your applications with a particular library, you simply add the necessary *-lib* switch to your *hxml* file.

```
-main MyApp
-lib nGui
-neko app.n
```
The nGui Library

As previously mentioned, the nGui library is easy to pick up and run with, so this will be the library of choice to get your feet wet. At the time of writing this book, the nGui library provides bindings to around 30 percent of the controls available from the IUP library version 2.6, supporting most of the bare essentials for the majority of applications, but should support nearly all of the IUP libraries when this book goes to print. Table 17-1 details the nGui control list.

Table 17-1

<table>
<thead>
<tr>
<th>Control</th>
<th>Functionality</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>Standard button control</td>
<td>Interactive</td>
</tr>
<tr>
<td>Label</td>
<td>Standard label control</td>
<td>Informative</td>
</tr>
<tr>
<td>List</td>
<td>List, drop-down list, and combo box functionality</td>
<td>Interactive</td>
</tr>
<tr>
<td>OLE</td>
<td>Container for ActiveX controls (Windows only)</td>
<td>External control support</td>
</tr>
<tr>
<td>Text</td>
<td>Single and multiline text field control</td>
<td>Interactive</td>
</tr>
<tr>
<td>Radio Buttons</td>
<td>Standard radio buttons</td>
<td>Interactive</td>
</tr>
<tr>
<td>Menu</td>
<td>Dialog-based and context-sensitive menu control</td>
<td>Interactive</td>
</tr>
<tr>
<td>Dialog</td>
<td>Standard dialog control</td>
<td>Container</td>
</tr>
<tr>
<td>Frame</td>
<td>Layout container with label</td>
<td>Layout / Informative</td>
</tr>
<tr>
<td>HBox</td>
<td>Layout container</td>
<td>Layout</td>
</tr>
<tr>
<td>VBox</td>
<td>Layout container</td>
<td>Layout</td>
</tr>
<tr>
<td>Tabs</td>
<td>Multipage layout container</td>
<td>Layout</td>
</tr>
</tbody>
</table>

Unfortunately, the IUP library doesn’t currently support a checkbox control, but facilitates the same functionality through the use of a toggle button control, instead.

The nGui Manager Class

The nGui library follows the same basic principals of any event-based programming architecture, whereby the application presenting the interface will enter a perpetual loop and should only exit the loop upon the applications exit request. While within the loop, all communications between the application business logic and the GUI controls are handled via callbacks that are triggered when real-time events occur. These events can be anything from a button click to a change of selected item within a list control, but can also include a defined criteria match within the operating system’s current state, such as an alarm when a particular clock time has been met. nGui handles the events loop internally, so no custom business logic may exist within the loop. Instead, one must delineate functions that will be assigned to the various controls in existence within the application through predefined hooks. When the events associated with the hooks are triggered, the functions are then executed.
In nGui, the primary loop, as well as any required initialization and cleanup routines, are detailed in the Manager class. Normally, when using nGui, the methods of the Manager class will be the first and last functions you call within the initial application method. The Manager class also includes several functions that deal with dialogs or controls on the whole. Table 17-2 details the methods of the Manager class.

### Table 17-2

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new() : Void</td>
<td>Initializes the nGui module.</td>
</tr>
<tr>
<td>close() : Void</td>
<td>Cleans up after the nGui framework, freeing any acquired memory.</td>
</tr>
<tr>
<td>loop() : Void</td>
<td>Starts the main application event loop.</td>
</tr>
<tr>
<td>getText( title : String, text : String ) : Void</td>
<td>Displays a text retrieval dialog with default text value and OK and Cancel buttons.</td>
</tr>
<tr>
<td>getFile(filter : String ) : Void</td>
<td>Displays a file dialog with extension filter control.</td>
</tr>
<tr>
<td>alarm( title : String, message : String, button1 : String, button2 : String, button3 : String ) : Void</td>
<td>Displays a message dialog with choice of one, two, or three buttons.</td>
</tr>
<tr>
<td>appendItem( container : Void, item : Void ) : Void</td>
<td>Appends the object specified by param one into the object specified by param two.</td>
</tr>
<tr>
<td>getScreenSize() : Void</td>
<td>Returns an array of integers depicting the width and height of the desktop.</td>
</tr>
</tbody>
</table>

The important methods to take note of are the top three methods: the constructor, loop, and close. As an example, then, the barest minimum that you could enter into a script utilizing the nGui framework might look like this:

```neko
import nGui.Manager;

class Basic {
    public static function main() {
        var bs = new Basic();
    }

    public function new() {
        var mng = new Manager();
        mng.alarm( "Welcome...", "Welcome to the nGui framework", "Okay" );
        mng.loop();
        mng.close();
    }
}
```
Part III: Extending the Possibilities

Now, you might think that this could be simplified further still, by removing the call to the `alarm` method of the `Manager` class. However, the `nGui` loop only remains in existence as long as controls exist in play. Therefore, if no control is called at all, then the loop will immediately exit.

**Working with Static Controls**

Static controls provide no interactive functionality, but exist to provide an aesthetic or informative role. For example, the label control cannot be interacted with in any way, yet its value may be changed and updated in real time, thus providing information to the user. You’ll look at using this control first, as it doesn’t support events:

```java
import nGui.Manager;
import nGui.controls.containers.VBox;
import nGui.controls.containers.Dialog;
import nGui.controls.Label;

class Static
{
  public static function main()
  {
    var bs = new Static();
  }
  public function new()
  {
    var mng = new Manager();
    var vbox = new VBox();
    var lbl1 = new Label( "This is the nGui label control" );
    var lbl2 = new Label( "This is aligned right" );
    var lbl3 = new Label( "" );
    var lbl4 = new Label( "This is inactive." );
    lbl2.align = "RIGHT";
    lbl2.expand = true;
    lbl4.active = false;
    lbl3.setHSeparator();
    vbox.appendChild( lbl1 );
    vbox.appendChild( lbl2 );
    vbox.appendChild( lbl3 );
    vbox.appendChild( lbl4 );
    var win : Dialog = new Dialog( vbox );
    win.show();
    mng.loop();
    mng.close();
  }
}
```

If you run this script now, you should be presented with a small window containing several lines of text and a horizontal divider, as shown in Figure 17-1. Quite a lot is going on here that really only stems from three new controls: `Label`, `VBox`, and `Dialog`. You’ll look at the `VBox` and `Dialog` controls in detail later in this chapter. The `Label` control performs several functions in this example, both a textual control and a graphical divider. The `Label` control can also act as a display for images and provide a vertical divider, though this is not shown in the example. This is a common feature of `nGui`, whereby a control can serve numerous functions depending on its context or how it is configured. Table 17-3 details the methods of the `Label` control, while Table 17-4 details its properties.
Chapter 17: Desktop Applications with Neko

The Control Class

The nGui class framework is based on an object-oriented model, whereby all controls, including the Label control, inherit a base class called Control. The Control class provides a number of universal functions used for control manipulation and for setting properties that are available to all controls in nGui, as well as permitting a generic type for the controls. Any of the controls can make use of the attributes as detailed in the Control class, though not all of the controls will necessarily display any obvious change based on those attributes. Table 17-5 details the attributes contained in the Control class.

Table 17-3

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new( text : String ) : Void</td>
<td>Acquires the textual value to initialize the Label control.</td>
</tr>
<tr>
<td>setHSeparator() : Void</td>
<td>Converts the Label control to a horizontal divider.</td>
</tr>
<tr>
<td>setVSeparator() : Void</td>
<td>Converts the Label control to a vertical divider.</td>
</tr>
</tbody>
</table>

Table 17-4

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>align : String</td>
<td>Determines the alignment of the text — RIGHT, CENTER, or LEFT.</td>
</tr>
<tr>
<td>text : String</td>
<td>Sets and gets the textual value of the Label control.</td>
</tr>
</tbody>
</table>

Table 17-5

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bgColor : Int</td>
<td>Sets the background color for the control.</td>
</tr>
<tr>
<td>fgColor : Int</td>
<td>Sets the foreground color for the control.</td>
</tr>
<tr>
<td>active : Bool</td>
<td>Determines its aesthetic state as either greyed out (inactive) or full (active).</td>
</tr>
</tbody>
</table>

Table continued on following page
Part III: Extending the Possibilities

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expand : Bool</td>
<td>Expands the width and/or height bounds of the control to its parent width and/or height.</td>
</tr>
<tr>
<td>width : Int</td>
<td>Determines the width of the control.</td>
</tr>
<tr>
<td>height : Int</td>
<td>Determines the height of the control.</td>
</tr>
<tr>
<td>title : String</td>
<td>Element’s title. It is often used to modify some static text of the element.</td>
</tr>
<tr>
<td>value : String</td>
<td>Often used to change a control’s main value.</td>
</tr>
<tr>
<td>tip : String</td>
<td>Determines the tooltip for the control.</td>
</tr>
<tr>
<td>visible : Bool</td>
<td>Determines the visibility of the control.</td>
</tr>
</tbody>
</table>

Thanks to the universal attributes as laid out in the `Control` class, you can now rewrite the previous example to include many possible alterations to the `Label` control. For example, you could modify the color of the control and provide a tooltip when the mouse arrow is hovering over the control:

```java
import nGui.Manager;
import nGui.controls.containers.VBox;
import nGui.controls.containers.Dialog;
import nGui.controls.Label;

class StaticColor
{
    public static function main()
    {
        var bs = new StaticColor();
    }
    public function new()
    {
        var mng = new Manager();
        var vbox = new VBox();
        var lbl1 = new Label( "This is the nGui label control" );
        var lbl2 = new Label( "How is this label aligned?" );
        var lbl3 = new Label( "" );
        var lbl4 = new Label( "This is inactive" );
        lbl1.fgColor = 0xFFFF00;
        lbl1.bgColor = 0x0000FF;
        lbl2.align = "RIGHT";
        lbl2.expand = true;
        lbl2.fgColor = 0xFF0000;
        lbl2.bgColor = 0x00FFFF;
        lbl2.tip = "To the right, of course!";
        lbl4.active = false;
        lbl3.fgColor = 0x00FF00;
        lbl3.bgColor = 0xFFFF00;
        lbl3.setHSeparator();
        lbl4.fgColor = 0x0000FF;
        lbl4.bgColor = 0x00FFFF;
    }
}```
vbox.appendChild( lb1 );
vbox.appendChild( lb2 );
vbox.appendChild( lb3 );
vbox.appendChild( lb4 );
var win : Dialog = new Dialog( vbox );
win.show();
mng.loop();
mng.close();
}

**Container Classes**

Aside from utility dialogs, such as the file selector and message box, the nGui library provides only a single class, called Dialog, for displaying controls. This dialog represents a standard window object, but can be customized to suit the style of window you require, providing such alterations are set before the window is first shown.

```javascript
var dialog = new Dialog( control );
```

Upon creation of the dialog, an object must be passed as a parameter. This object will then be added to the dialog display and set as a child of that dialog. It is possible to add further controls to the dialog by using the appendChild method of the Dialog class. However, under normal circumstances, only one child will be appended: a container object. The reason for this is that the Dialog control does not support any layout capabilities, so any objects you append to the dialog will simply line up one by one horizontally. To remedy this, nGui provides several controls that do little else than organize the physical positioning of controls. These positioning controls are themselves nestable objects; thus, they can be combined to provide a desired look and feel when developing your GUI.

The layout container controls provided by nGui do not support absolute positioning of controls. The reason for this is to facilitate resizing of the parent dialog or container object. By allowing controls to position themselves based on alignment and padding, it should theoretically be easier to create a GUI that looks pretty much the same regardless of the dimensions of the parent dialog.

Table 17-6 details the layout controls provided by the nGui framework.

**Table 17-6**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fill</td>
<td>Creates a container that dynamically fills empty spaces.</td>
</tr>
<tr>
<td>HBox</td>
<td>Creates a container that lays out its child objects side by side in a horizontal fashion.</td>
</tr>
<tr>
<td>VBox</td>
<td>Creates a container that lays out its child objects side by side in a vertical fashion.</td>
</tr>
</tbody>
</table>

When using the layout container controls, you first need to work out how you would prefer the layout of your GUI to appear. Once this has been decided, you then need to consider which formation would yield the required results. This can sometimes be trial and error, as you can never truly guarantee that your
controls will sit where you would like them. For example, assuming you require a GUI containing three rows of controls, each of which consists of a label on the left-hand side and a text field on the right, you would most likely opt for one of the two layouts shown in Figure 17-2.

As you can see, both layouts provide the same rough positioning for child controls. However, as controls are rarely of the same dimensions, it's possible that grouped controls may not sit perfectly alongside one another where you'd most like them. In the previous example, it would be more important for each label control to sit vertically level with its paired text field. This is more likely to happen in the layout on the right, where the HBox controls enforce vertical alignment, albeit with a possible misalignment between rows for the left-hand side of each text field. In this circumstance, however, this can be rectified by setting a width parameter for the Label controls, thus providing no reason for the divider between cells to deviate.

**Working with Interactive Controls and Events**

The interactive controls in nGui include buttons, text fields, combo boxes, menus, and radio buttons. Each of these controls extends the Control class and exists to provide the functionality of your GUI. The major differing factor between interactive and static controls is that the interactive controls fire events when the user interacts with them. These events can then be handled in your code through the use of function pointers.

**The Button Control**

The most simple of the interactive controls is the Button control. Here, you are provided with a simple clickable object that can display either textual or graphical content. The Button control provides support for the single solitary event, onClick, which is fired both when the button is pressed and when it is released. When the event is triggered, the function assigned to the onClick event is called and passed an integer value depicting the current state of the button; 1 for pressed and 0 for released. The following example demonstrates this characteristic:

```java
import nGui.Manager;
import nGui.controls.Label;
import nGui.controls.Button;
import nGui.controls.containers.VBox;
import nGui.controls.containers.Dialog;

class ButtonSampleCallback
```
public static function main()
{
    var bs = new ButtonSampleCallback();
}

public function new()
{
    var mng = new Manager();
    var vert : VBox = new VBox();
    var lbl : Label = new Label( "Click the button below" );
    var button1 : Button = new Button( "Click Me" );
    vert.appendChild( lbl );
    vert.appendChild( button1 );
    button1.onClick = function( val : Int )
    {
        lbl.text = "Button pressed = " + val;
        return null;
    }
    var wind : Dialog = new Dialog( vert );
    wind.title = "Buttons Sample";
    wind.show();
    mng.loop();
}

The example should display a dialog, similar to the one detailed in Figure 17-3. Upon clicking the button control, the onClick event for button1 is fired and the label lbl is updated to show the state of the button.

Figure 17-3

**The Text Control**

Just as the Label control facilitates multiple guises, so the Text control also offers more than one display and can be rendered as either a single-line text field or a multiline field. This is decided when the Text control is instanced, by passing a Boolean value as the constructor parameter.

Like the Button control, the Text control also provides one event handler, which is fired whenever the content of the field is changed. The handler for this event accepts two parameters. The first parameter is the ASCII (American Standard Code for International Interchange) code of the last key pressed, while the second parameter represents the content of the field should the value be accepted:

```text
textField.onChange = function( code : Int, content : String ) : Int
{
    neko.Lib.println( "The key \" + code + \" was added to \" + content + \"\" );
}
```
Part III: Extending the Possibilities

The return code of the onChange event can be either an integer representation of a ReturnType enumerator value or the ASCII value of a key you wish rendered instead. The ReturnType enumerator provides four values representing the type of value returnable from an nGui callback. These are DEFAULT, CLOSE, CONTINUE, and IGNORE. Normally, DEFAULT and IGNORE perform the same function, except under special circumstances, while the CLOSE value causes the application loop to end, and CONTINUE passes the event to the parent control.

As ReturnType cannot automatically be converted to an integer value, the Control class provides the function RTToInt. This method must be called if a ReturnType enumerator is used. Here is an example of how the Text control might be used:

```java
import nGui.Manager;
import nGui.controls.Text;
import nGui.controls.Control;
import nGui.controls.ReturnType;
import nGui.controls.containers.VBox;
import nGui.controls.containers.Dialog;
class TextSample
{
    public static function main()
    {
        var bs = new TextSample();
    }
    public function new()
    {
        var mng = new Manager();
        var vert : VBox = new VBox();
        var txt1 : Text = new Text( false );
        txt1.readonly = true;
        txt1.width = 150;
        var txt2 : Text = new Text( true );
        txt2.setSize( "150", "50" );
        vert.appendChild( txt1 );
        vert.appendChild( txt2 );
        txt2.onChange = function( char : Int, val : String )
        {
            txt1.value = "Button pressed = " + Std.chr( char );
            return txt2.RTToInt( DEFAULT );
        }
        var wind : Dialog = new Dialog( vert );
        wind.title = "Text Sample";
        wind.show();
        mng.loop();
    }
}
```

This example should produce a dialog similar to that shown in Figure 17-4. The Text control provides several properties and functions you can use to interact with the control. The properties are detailed in Table 17-7 and the functions in Table 17-8.
Chapter 17: Desktop Applications with Neko

Table 17-7

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cursor : Int</td>
<td>Specifies the current position of the insertion point.</td>
</tr>
<tr>
<td>maxChars : Int</td>
<td>Specifies the maximum number of characters allowed in the control. The default maximum number of characters is $2^{31}$.</td>
</tr>
<tr>
<td>readOnly : Bool</td>
<td>Allows the user to read text without being able to change it. The default is false.</td>
</tr>
<tr>
<td>border : Bool</td>
<td>Shows a border around the control. The default is true.</td>
</tr>
</tbody>
</table>

Table 17-8

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append(val : String)</td>
<td>Appends text to the end of the field’s current content.</td>
</tr>
<tr>
<td>insert(val : String)</td>
<td>Inserts text at the current cursor location.</td>
</tr>
</tbody>
</table>

The List Control

The List control provides the functionality of five different, albeit related, controls: a standard single selection list, an editable selection list, a multiple selection list, a drop-down control, and an editable combo box control. The default control is the single selection list, though the type of control supplied can be changed, prior to its parent dialog becoming visible, by manipulating the control’s properties.

Once the control has been instantiated, the items populated within the list must then be added one at a time. You do this using the `addItem` method of the `List` class by passing it a string value representing the text to be displayed for the item. This can be done as many times as there are items to be added, and can be called even at run time when the list is visible.

The List control provides a single callback, which is called when an initial item is selected within the list. This callback is not called on subsequent selections, however, when the list object is set as a multiple
Part III: Extending the Possibilities

selection list, though will still be called should the selected items be cleared and a new initial item selected. At any time, the selected item details can be retrieved through querying the `selectedItems` property. The returned value and type is dependent upon the type of the list. Table 17-9 details the values returned by the `selectedItems` property for the `List` control, based on its characterization.

**Table 17-9**

<table>
<thead>
<tr>
<th>List Type</th>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single selection list or drop-down list</td>
<td>Int</td>
<td>Index of the selected item; starts at 0</td>
</tr>
<tr>
<td>Editable single selection list</td>
<td>String</td>
<td>The currently entered text, or the string representation of the last selected item</td>
</tr>
<tr>
<td>Multiple selection list</td>
<td>Array&lt;Bool&gt;</td>
<td>An array of Booleans depicting the selection state of each item in the List</td>
</tr>
<tr>
<td>Combo box (editable) list</td>
<td>String</td>
<td>The currently entered text, or the string representation of the last selected item</td>
</tr>
</tbody>
</table>

At present, the `nGui List` control provides only a subset of the functionality offered by the IUP library `List` control. This may change over time as the haXe community increases its uptake of the library. However, it is the purpose of `nGui` to maintain a simple interface for rapid applications development, so any additions will be minimal, and will certainly maintain its current feature set.

The following example highlights the simplicity of the `List` control for complex functionality:

```haXe
import neko.Lib;
import nGui.Manager;
import nGui.controls.List;
import nGui.controls.Label;
import nGui.controls.Button;
import nGui.controls.Control;
import nGui.controls.containers.VBox;
import nGui.controls.containers.HBox;
import nGui.controls.containers.Dialog;
class ListSample
{
    var t : Int;
    public static function main()
    {
        var a = new ListSample();
    }
    public function new()
    {
        var mng = new Manager();
        var vert : VBox = new VBox();
        var horz : HBox = new HBox();
        var lbl : Label = new Label( "No items selected" );
    }
}
```
Chapter 17: Desktop Applications with Neko

```javascript
var lst : nGui.controls.List = new nGui.controls.List();
var lst2 : nGui.controls.List = new nGui.controls.List();
var lst3 : nGui.controls.List = new nGui.controls.List();
var lst4 : nGui.controls.List = new nGui.controls.List();
var lst5 : nGui.controls.List = new nGui.controls.List();

for ( i in 0...10 )
{
    lst.appendItem( "item " + ( i + 1 ) );
    lst2.appendItem( "item " + ( i + 1 ) );
    lst3.appendItem( "item " + ( i + 1 ) );
    lst4.appendItem( "item " + ( i + 1 ) );
    lst5.appendItem( "item " + ( i + 1 ) );
}

vert.appendChild( lbl );
vert.appendChild( horz );
horz.appendChild( lst );
horz.appendChild( lst2 );
horz.appendChild( lst3 );
horz.appendChild( lst4 );
horz.appendChild( lst5 );

lst2.multiple = true;
lst3.editbox = true;
lst4.dropdown = true;
lst5.dropdown = true;
lst5.editbox = true;
lst.onChange = function()
{
    lbl.text = Std.string( lst.selectedItems );
}

lst2.onChange = function()
{
    lbl.text = Std.string( lst2.selectedItems );
}

lst3.onChange = function()
{
    lbl.text = Std.string( lst3.selectedItems );
}

lst4.onChange = function()
{
    lbl.text = Std.string( lst4.selectedItems );
}

lst5.onChange = function()
{
    lbl.text = Std.string( lst5.selectedItems );
}

var wind : Dialog = new Dialog( vert );
w wind.title = "List Sample";
w wind.show();
mng.loop();
```

Part III: Extending the Possibilities

Figure 17-5 shows how this example might look on your computer.

![List Sample](image)

**Figure 17-5**

This example is really much longer than it needs to be, but at least shows, in the simplest form, all five `List` possibilities in action following a four step initialization process:

- Instantiate the `List`.
- Append the child item values.
- Set its properties, such as `multiple`, `dropdown`, and `editbox`.
- Set its callback handlers.

Indeed, the last two points here are optional, as the `List` does provide default property values, and it is quite acceptable to supply no callback handler whatsoever, but to rely on another control’s event in order to access the list’s selected items.

To finalize, the `List` control supports other properties not yet mentioned. These aren’t discussed, as they should be greatly relevant to list controls of other GUI libraries for other languages. However, for completion, you can review these properties in Table 17-10.

### Table 17-10

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>multiple</code> : Bool</td>
<td>Sets the type of <code>List</code> control from drop-down to standard list.</td>
</tr>
<tr>
<td><code>editbox</code> : Bool</td>
<td>Sets a drop-down control to combo box or standard list to editable list. It is not available on multiple selection lists.</td>
</tr>
<tr>
<td><code>dropdown</code> : Bool</td>
<td>Converts the initial <code>List</code> control to a drop-down list.</td>
</tr>
<tr>
<td><code>selectedItems</code> : Dynamic</td>
<td>Returns values depicting the currently selected item or items, as detailed in Table 17-9.</td>
</tr>
</tbody>
</table>
The Tab Control

While technically a container, the Tab control provides great functionality by maximizing space within a dialog where controls can be grouped by purpose. The Tab control, otherwise known as a Notebook in some GUI frameworks, is an excellent means of creating overlaying panels in a tabbed book format, so that only one set of grouped controls are visible at any one time, while at the same time, allowing all controls across all pages to be accessible at once.

The Tab control functions very similarly to a dialog control, whereby containers, rather than controls, are added as the child object. Then, for each container appended to the Tab control, a new page is added to facilitate it. Once all pages are added, you can then name the tabs associated with the pages by passing a string to the setTitle method of each container. setTitle is a method of the base Control class, so all containers and controls, including the Tab control itself, can make use of it.

Unlike other containers, the Tab control requires all child Control objects to be appended at the same time when the Tab control is instantiated. You do this by storing these objects into an array of type Control, then passing the array as a parameter to the instantiation. You cannot append further objects to the Tab control itself once the Tab control is instantiated.

```javascript
var tabControl = new Tabs( objectsArray );
```

The Tab control provides the single event, onChange. This event is called whenever a new tab is clicked and is passed two parameters; the first is the title of the newly selected tab and the second is the title of the old tab. The event function expects an integer return value, though you should derive this from the appropriate ReturnType enumerator. Therefore, if you return IGNORE, the old tab should remain selected.

The following is an example of the Tab control in action. Figure 17-6 shows how this example might appear when run.

```javascript
import nGui.controls.Button;
import nGui.controls.Control;
import nGui.Manager;
import nGui.controls.containers.Dialog;
import nGui.controls.containers.Tabs;
import nGui.controls.ReturnType;

(continued)
Part III: Extending the Possibilities

(continued)

class TabbedSample
{
    public static function main()
    {
        var bs = new TabbedSample();
    }
    public function new()
    {
        var mng = new Manager();
        var button1 : Button = new Button( "Button A" );
        var button2 : Button = new Button( "Button B" );
        var button3 : Button = new Button( "Button C" );
        var arr : Array<Control> = new Array();
        arr.push( button1 );
        arr.push( button2 );
        arr.push( button3 );
        button1.tabTitle( "One" );
        button2.tabTitle( "Two" );
        button3.tabTitle( "Three" );
        var tabs : Tabs = new Tabs( arr );
        tabs.position = "BOTTOM";
        tabs.onChange = function ( newTab : String, oldTab : String )
        {
            neko.Lib.println( "The new tab is : " + newTab );
            neko.Lib.println( "The old tab is : " + oldTab );
            return tabs.RTToInt( DEFAULT );
        }
        var wind : Dialog = new Dialog( tabs );
        wind.setSize( "200", "40" );
        wind.title = "Tabbed Sample";
        wind.show();
        mng.loop();
    }
}

Working with Menus

The final control available in nGui for discussion is the Menu control. Menus in nGui are only applicable to the Dialog control, and at present, there is no support for context menus relative to individual controls. However, this may change as the nGui library matures.
The menu package in nGui consists of three classes and an interface. The three classes are Menu, Item, and Submenu. Menu is the parent governing control in the menu package representing the entire menu functionality, Item is the child class which represents each clickable label within any given menu display, while Submenu represents any Item's grouped by context. The interface, IMenu, is required to provide an interchangeable association between menu items, represented by the Item class, and submenu items represented by the Submenu class.

Creating a Menu

To create a menu, you must first instantiate the Menu class, passing it an identifier string. This string can be any text you like, but will later be used to assign the menu to the dialog, so it should generally be something appropriate.

```javascript
var menu = new Menu("Menu Identifier");
```

Once the Menu class is instantiated, you can then start adding Items. If the containing Menu object is the parent in the menu hierarchy, then any added Items will show across the top of the Dialog. Likewise, if the menu has been assigned to a Submenu object, then the added Items will be displayed within a pop-up container. You add Items to a Menu object using its addItem method, as shown in the following example:

```javascript
menu.addItem( myItem );
```

A separator can also be added in this fashion by assigning two hyphens (--) as the label of the item. Under such circumstances, however, the methods of the Item class will no longer apply to the item.

Appending Submenus

The addItem method of the Menu class accepts objects that implement the IMenu interface. This means that both objects of the Item class and those of the Submenu class can be added to an object of type Menu. A submenu is essentially an object consisting of a child menu object and a reference string. The submenu object then acts as a proxy for the child menu object, so it can be safely added to a parent menu. You create an object of type Submenu by passing both menu and reference string to its constructor, like this:

```javascript
var sub = new Submenu( "my menu", menu );
```

In this instance, the sub variable can then be used in the same way as an Item object, where the reference string is used as the items label.

Menu Events

Objects derived from the Item class support the event onClick. This event will usually need to be added to each item object individually as the event function accepts no parameters, so it is not often possible to obtain which menu item was selected. The following example shows the menu framework in action and illustrates how the menu events might be used. Figure 17-7 then shows how this example should look.

```javascript
import nGui.Manager;
import nGui.controls.Label;
import nGui.controls.menu.Menu;
```

(continued)
import nGui.controls.menu.IMenu;
import nGui.controls.menu.Submenu;
import nGui.controls.menu.Item;
import nGui.controls.containers.Dialog;
class MenuSample
{
    private var mnu : Menu;
    private var smnu : Menu;
    private var s1 : Submenu;
    public static function main()
    {
        var bs = new MenuSample();
    }
    public function new()
    {
        var mng : Manager = new Manager();
        mnu = new Menu("MyMenu");
        smnu = new Menu("DDMenu");
        var i1, i2, i3, i4 : Item;
        i1 = new Item("Item 1");
        i2 = new Item("Item 2");
        i3 = new Item("--");
        i4 = new Item("Item 3");
        smnu.addItem(i1);
        smnu.addItem(i2);
        smnu.addItem(i3);
        smnu.addItem(i4);
        i1.onClick = function()
        {
            neko.Lib.println("item 1 was clicked");
        }
        i2.onClick = function()
        {
            neko.Lib.println("item 2 was clicked");
        }
        i4.onClick = function()
        {
            neko.Lib.println("item 3 was clicked");
        }
        s1 = new Submenu("Submenu", smnu);
        mnu.addItem(s1);
        var lbl : Label = new Label("You should see the menu above");
        var wind : Dialog = new Dialog(lbl);
        wind.menu = "MyMenu";
        wind.setSize("200", "200");
        wind.title = "Menu Sample";
        wind.show();
        mng.loop();
        mng.close();
    }
}
Chapter 17: Desktop Applications with Neko

The hxGtk Library

The hxGtk library functions a little differently to the nGui library. Indeed, to contemplate using hxGtk, it is advised that you keep a reference to the GTK library handy, as well as decent SQLite management software, as you’ll need these to decipher what functions are available to you and how to use them.

When Ritchie Turner developed the bindings for hxGtk, he decided that, having been faced with the impossible task of wrapping such a vast library, it would pay to develop a script that would produce most of the work for him based on the source of GTK. So, Ritchie wrote a generator in Ruby, which produced a C file containing over 2300 functions. Now, as you know, the C source of a Neko library is only half of the undertaking. The rest of the work involves creating a haXe framework that will allow any user of the library to access its functionality in a friendly object-oriented way. Knowing this, Ritchie avoided creating classes to contain all 2300 of his newly created functions and instead devised a proxy system that allows dynamic access to the functions on an as needed basis. This meant that very few of the functions were then hard coded into the haXe classes and were instead stored within a SQLite database file to provide a safe lookup facility.

Okay, so where does this leave the developer wishing to use hxGtk? Well, the long-term plan for the hxGtk library is to eventually create all of the classes necessary to wrap the entire GTK library. Indeed, a certain amount of this class framework already exists, but at the time of writing this book, it is still very young and will likely change frequently before a finalized framework is available. Therefore, until such a time that hxGtk is considered finished, one would need to learn how the proxy system operates in order to use any part of hxGtk. Now, this isn’t as bad as it sounds. The benefits of having the proxy system are that the entire GTK library is available to the haXe community, today. Also, all of this functionality can be accessed in the same manner, so once you learn how to use it, you need not fear any unusual nuances within the library, except for those that exist in the GTK library itself.

The hxGtk Function Proxy

Now, before you begin, you should repeat several times the words “the hxGtk proxy is my friend”. You can say this about ten times or so, or at least long enough that you don’t automatically skip this part of the book in terror. Once you understand the basics of the proxy system, you’ll wonder why all colossal widget libraries don’t use a similar system.

If you own SQLite management software, use it to open the file gtk.db within the hxGtk installation directory. Within this file, you’ll find a single table called “functions” containing four columns. These
Part III: Extending the Possibilities

columns detail the row identity, the name of the function, the number of parameters accepted by the function, and the type of its return value, in that order. Start your journey into using hxGtk by instantiating a control using the data stored in this table. This way, you will see that pretty much every control within the hxGtk library can be treated in the same fashion, and therefore, there is little of the library that should be overwhelming, despite its astronomical size.

The function names stored in the neko_name column are named after the GTK library equivalents. For the most part, the differences between the function names in hxGtk and those of GTK are that the hxGtk function names have had any underscores (_) removed, and have been prepended with the characters hx_. Knowing this should greatly aid function usage.

Looking at the records stored in the SQLite file, locate the one with the name hx_gtkwindownew. As you can probably guess, this function provides the constructor functionality for the window object. If you guessed incorrectly, take a closer look at the function name. You’ll see that the name is made up of three segments. The first of these segments is always the word hx_gtk. This is a formality, and was necessary only to provide a namespace for the function collective within the C source code. The second part of the name is window. This part represents the object that the function is associated with. You could imagine this as being a class that the function is a member of, if such a class existed. Finally, the third part is the name of the action or method that will be applied to the object. In this case, the method is new, which is a constructor method. With this in mind, use the function to instantiate a new window object.

When you use the proxy system, all functions of hxGtk are called using the static call function of the Gtk class. This function reforms the passed function name, then performs a checkup on the SQLite database file to check if it exists. If the function does exist, then a link to the function in C is created and executed, along with any parameters. A call to the call function would look a little like this:

```javascript
var wnd = Gtk.call.windowNew([TOPLEVEL]);
```

The parameters in this function call are passed within an array construct. This way, you can pass as many parameters as are required by the called function, and Neko will deal with it appropriately. In the case of the preceding example, the passed parameter is a value of the WindowType enumerator. You can find this, and other enumerators for other functions, within the Gtk class. If you find yourself in a position where an enumerator for a particular object is not available, feel free to add it to the class yourself and support the hxGtk user community.

As you can see in the example, the function you’re calling is called windowNew. The haXe language uses this function name and resolves it within a generic function built to handle any name appended to Gtk.call. In this instance, the appended name is converted to lowercase and has the string hx_gtk prepended to the beginning of it, which will mean that, in the current circumstance, windowNew will resolve to hx_gtkwindownew.

That’s all there is to it. The resolved function call may or may not return a value. This is detailed in the SQLite database file under the rettype column. When creating a new control, this return value will likely be a C pointer handle to the control, so you must store it in a variable of type Void, so that Neko knows it is not a type it can modify. Alternatively, the type may be one of the values listed in Table 17-11.
Chapter 17: Desktop Applications with Neko

Table 17-11

<table>
<thead>
<tr>
<th>hxGtk Type</th>
<th>haXe Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gunichar, gchar</td>
<td>String</td>
<td>Unicode and standard character strings.</td>
</tr>
<tr>
<td>gboolean</td>
<td>Bool</td>
<td>A standard Boolean. Either true, or false.</td>
</tr>
<tr>
<td>gulong, guint32,</td>
<td>Int</td>
<td>Various sized integers. These will be converted to a</td>
</tr>
<tr>
<td>guint16, guint,</td>
<td></td>
<td>haXe Int, though under some circumstances, data</td>
</tr>
<tr>
<td>guint, gint</td>
<td></td>
<td>truncation may occur.</td>
</tr>
<tr>
<td>gfloat, gdouble</td>
<td>Float</td>
<td>Various sized floating-point values. Unlike integers,</td>
</tr>
<tr>
<td>[other]</td>
<td>Void</td>
<td>no truncation will occur.</td>
</tr>
</tbody>
</table>

The Gtk Class

You now know how to execute the various functions within the hxGtk ndll library. However, aside from control instantiation and interaction, the actual running of hxGtk can be far simpler and certainly less generic, thanks to the Gtk class. As with nGui, the hxGtk library needs to first be initialized. Likewise, once your controls have been created and configured, the application then needs to enter an event loop, so that messages passed from the operating system can be handled. All of this can be accomplished with the methods contained in the Gtk class.

As you have already seen this in action with the nGui library, you can jump straight into this, now, with the help of an example. An image of this is shown in Figure 17-8.

```javascript
import gtk.Gtk;
class Basic {
  public static function main() {
    Gtk.init();
    var win = Gtk.call.windowNew([TOPLEVEL]);
    Gtk.call.windowSetTitle([win, "Basic Example"]);
    Gtk.call.windowPresent([win]);
    Gtk.run();
  }
}
```
Part III: Extending the Possibilities

**hxGtk Events**

In such a short space, you’ve already covered a great deal. With the little you’ve learned over the last three pages, and the help of the GTK online documentation, you should be able to piece together any of the library’s available controls. All that remains, now, is to interact with the controls through the use of events and event handlers.

In GTK, events are known as signals, and as *hxGtk* is a generated wrapper for GTK, the term is carried throughout. Handling events in *hxGtk* is very similar to how events are handled in *nGui*, with the exception that all event handlers in *hxGtk* follow the same parameter signature. In *nGui*, the parameters accepted by an event handler vary from control to control, so it is necessary to know what values are returned in order to properly construct the handler function. *hxGtk* does this differently by accepting a **Dynamic** value, which essentially provides a container for the return values, regardless of what they might be. The parameter container looks like this:

```
{sender : Dynamic, sig_types : Array<Dynamic>, sig_prms : Dynamic}
```

Here, the **sig_types** and **sig_prms** values are linked; **sig_types** provides an array of the names, in string value, for each item stored in the **sig_prms** object. If only one value is returned, **sig_prms** will be of the type of that value, whereas if more than one item is returned, **sig_prms** will be an array. Despite this, **sig_types** will always be an array of one or more items.

The third item in the returned event object is the **sender** property. This value is merely a returned reference to data that was passed into the event construction when it is initially set up. Now, normally, this value will be a pointer to the object that made the event, thus providing a simple means to work out who fired the event. However, one could quite literally store anything they liked into this parameter, so feel free to use it as you see fit.
Chapter 17: Desktop Applications with Neko

Following is an example of an event handler in use:

```xml
import gtk.Gtk;

class Events {
    public static var signal = neko.Lib.load("hxGtk","hxs_closure_connect",4);
    public static function main() {
        var events = new Events();
    }
    public function new() {
        Gtk.init();
        var win = Gtk.call.windowNew( [TOPLEVEL] );

        var func = function( event : Dynamic ) {
            neko.Lib.println( "sender : " + event.sender );
            var types : Array<Dynamic> = untyped Array.new1( event.sig_types,
                __dollar__asize( event.sig_types ) );
            var values : Array<Dynamic> = untyped Array.new1( event.sig_prms,
                __dollar__asize( event.sig_prms ) );
            for ( i in 0...types.length )
                neko.Lib.println( "parameter " + types[i] + " = " + values[i] );
        }
        var button = Gtk.call.ButtonNewWithLabel( [neko.Lib.haxeToNeko( "Button" )] );
        Gtk.call.containerAdd( [win, button] );
        Gtk.call.widgetShow( [button] );
        signal( button, neko.Lib.haxeToNeko( "clicked" ), func, this );
        Gtk.call.windowSetTitle( [win, "Basic Example"] );
        Gtk.call.widgetShow( [win] );
        Gtk.run();
    }
}
```

As you can see, the signal function that handles the event handling assignment derives from `hxs_closure_connect` from the `hxGtk` ndll. This function call requires four parameters. The first of these is the GTK object, or widget if you like, that will fire the event, the second parameter is a Neko string detailing the type of event to register, while the other two required parameters include the function to call when the event is raised, and an object that will simply be carried over, within the event object, to the events function.

If you look closely at the lines within the custom event handling function, you can see where the data from the event parameter is extracted into arrays. Here, if the value of `sig_prms` is not an array, it is still packed into one, thus paving the way for consistency.

**Rapid hxGtk GUI Development with Glade**

One of the benefits of using GTK for GUI development is the Glade Integrated Development Environment (IDE). Glade is a cross-platform tool, and is very easy to use. Unfortunately, detailing how to use Glade in all its glory is beyond the scope of this book as it is very functional, so only the minimum
Part III: Extending the Possibilities

functionality required to use Glade in haXe will be shown, as well as how to use a Glade file within a haXe project using hxGtk, and how to interact with the controls it generates.

The Glade IDE

At the time of writing this book, the current version of Glade is 3.1 for UNIX-based systems and 3.0.2 for Windows systems. Version 3+ of Glade is a recent rewrite of the original Glade application, but also a welcome one, as it sports a new, more intuitive interface. You can download the latest version of Glade from http://gladewin32.sourceforge.net/ for Windows users and http://glade.gnome.org/ for Linux and Mac users.

Once you have Glade up and running, you should be presented with three dialogs. The primary dialog is the main project work area, where a hierarchy, listing the controls used in your project, is displayed. The other two dialogs present the properties and control palettes. Figure 17-9 illustrates how these dialogs might look on your system.

To start using Glade, create a new project now by choosing File ‹ New from the main menu at the top of the primary dialog. This ensures that you are starting from a clean slate.

Next, click the window icon, shown on the control palette beneath the Toplevels subheading. This places a new window control in the project control’s hierarchy and displays a new dialog representing the window control. From here, you can choose to modify the properties of this window using the properties dialog, if you want.
Chapter 17: Desktop Applications with Neko

To finalize your Glade example, click the button icon, and then click somewhere in the middle of the window. The button control should now appear within the window control, filling all available space. Figure 17-10 shows how this should now look.

![Figure 17-10](image)

Finish by saving your work as `test.glade` in the working directory. Glade files are runtime resource files, so you’ll need to include them with your application when distributed. You’ll get to actually instantiate this design, next.

**The Glade Class**

The Glade class is your portal to interacting with a Glade file. The actual structure and functionality of the Glade class is simple, yet it is extremely powerful as it leverages the Glade API within the GTK framework.

To use the Glade class, you simply instantiate it, passing as a parameter the filename of the Glade file you want to interact with:

```javascript
var glade = new Glade("test.glade");
```

Once acquired, the Glade API performs the task of building the GUI for you. You are then able to pick individual control references out of the Glade object for use as though you had instantiated them yourself. This is done using the `getObjHandler` method of the Glade object:

```javascript
var wnd = glade.getObj("window1");
```
Part III: Extending the Possibilities

Take a look at this now in an example:

```javascript
import gtk.Gtk;
import gtk.Glade;
class GladeTest
{
    public static var signal = neko.Lib.load("hxGtk","hxs_closure_connect",4);
    public static function main()
    {
        var gladetest = new GladeTest();
    }
    public function new()
    {
        Gtk.init();
        var glade = new Glade( "test.glade" );
        var win = glade.getObjHandle( "window1" );
        var button = glade.getObjHandle( "button1" );
        var func = function( event : Dynamic )
        {
            neko.Lib.println( "sender : " + event.sender );
            var types : Array<Dynamic> = untyped Array.new1( event.sig_types,
                __dollar_asize( event.sig_types ) );
            var values : Array<Dynamic> = untyped Array.new1( event.sig_prms,
                __dollar_asize( event.sig_prms ) );
            for ( i in 0...types.length )
                neko.Lib.println( "parameter " + types[i] + " = " + values[i] );
        }
        signal( button, neko.Lib.haxeToNeko( "clicked" ), func, this );
        Gtk.call.windowSetTitle( [win, "Glade Test"] );
        Gtk.call.widgetShow( [win] );
        Gtk.run();
    }
}
```

As you can see, using Glade can take out the whole monstrous monotony of configuring a GUI look and feel from your code, while leaving a very quick and simple means to add the necessary logic.

Summary

In many ways, you have only touched on the possibilities for GUI development in haXe. Indeed, when Xinf is finally production ready, there will probably be more powerful GUI creation capabilities than in any other scripting language, providing another notch to haXe’s belt. There is a lot that could not be covered in this chapter, mainly because of the infancy of the tools covered, but even in this early stage, haXe and Neko already now have the power to produce almost any application imaginable.

This chapter covered:

- What options are available to the desktop applications developer
- How to convert a compiled Neko script into an executable application
Chapter 17: Desktop Applications with Neko

- What Graphical User Interface (GUI) frameworks are available for haXe and Neko
- How to install the nGui and hxGtk libraries
- How to use the nGui and hxGtk libraries
- How to use Glade with the hxGtk library

In the next chapter, you look at building Flash-based GUI's using the SWHX library.
As you should now be aware, the Neko Virtual Machine provides an awesome engine for web applications, as well as desktop applications using the operating system’s native GUI libraries. The sheer flexibility of the Neko platform allows you, the developer, to put the Neko virtual machine to practically any task.

Now, as a haXe developer, which you can now safely consider yourself, you have at your disposal a means to produce the various required components of both web- and desktop-based applications through the one language. This in itself is a great boon, and will enable you to greatly reduce development time within your projects, while at the same time, allowing you to greatly increase your capabilities as you hone your knowledge of a single all-powerful language. The haXe compiler makes all of this possible, enabling you to fully carry out any development brief by supplying the necessary output formats and tying them all together. This is especially true of web-based applications, whereby the Flash, JavaScript, and Neko output can each be combined to form a single application, but it is also true of desktop applications, thanks to ScreenWeaver HX.

ScreenWeaver HX, or SWHX, provides the needed communications layer to enable a Neko application to display and manage a Flash-based graphical interface. You can liken this technology to SWF-to-Executable applications such as Multimedia’s Zinc or Northcode’s SWF Studio, only with a great increase in flexibility and a much faster, lighter framework.

Okay, so all you really need is Neko and the nGui library to create a desktop application, but it doesn’t give you the ultimate flexibility that both haXe and Neko were designed to provide. Besides, let’s face it, no operating system’s native graphical interface library could ever compare with the slick possibilities afforded by the Flash platform. What’s more, if a particular control required to perform a certain task is not available in the nGui library, or indeed any native GUI library, then using SWHX instead will afford you the option to create such a control yourself, giving you ultimate freedom and total satisfaction.
Part III: Extending the Possibilities

Now, enough with the sales talk, let's get down to business. In this chapter, you'll cover much of the SWHX library capabilities, including:

- Comparing SWHX to its non-Neko cousins
- Installing the SWHX library
- Initializing and utilizing the SWHX library
- Managing windows in SWHX
- Managing communications from Neko to Flash and back again
- Combining SWHX with dialogs from the SysTools library
- Working with menu's and system tray icons in SWHX

Why SWHX?

As haXe and Neko evangelists, we often have people ask us why they should use SWHX over commercial alternatives such as Multimedia's Zinc, Northcodes SWF Studio, or even Adobe's new AIR platform. Now, the issue at hand isn't really about whether SWHX is better than these alternatives, but where it fits in the grand scheme of things. Everything has its place, and usually, one application will often suit a purpose better than another. However, we find that SWHX ticks far more boxes, and fits snugly into far more holes than any other platform of its kind, anywhere.

Let's look at this more objectively. Zinc, SWF Studio, and AIR are known to be called SWF2Exe applications, because they convert an SWF into an executable format, hence SWF2Exe is SWF-to-Executable. This is something that the Flash IDE and player can already do in the form of a projector, but what a projector can't do is provide an entire framework of desktop-based functionality that can be accessed by the Flash movie. This is what Zinc, SWF Studio, and AIR do that make them invaluable. The only problem is, in order to provide such extra functionality, the executable structure with which the SWF Flash file must coexist has to provide an entire framework with the compiled application, as it doesn't know exactly what functionality the Flash SWF file might want to call. Zinc and SWF Studio achieve this by simply embedding the framework within the same executable file that contains the SWF, so that it can be ported from machine to machine much like any other standalone application. Adobe's AIR, on the other hand, takes a more arrogant approach by assuming that everyone who uses an AIR application will likely want to use many other AIR-based applications, so have packaged their entire framework into a runtime installation that must be carried out before any AIR programs can execute. We see this as the AIR platform's eventual undoing.

The approach taken by ScreenWeaver HX is different. So different, in fact, that it cannot be considered an SWF2Exe framework at all. By supplying a loose communications layer between a Neko application and an SWF, the whole compiled library theme is removed for something that is more pick and mix in nature. As you can choose to include or exclude any library you like from a Neko application, you can do exactly the same with one that utilizes SWHX, thus making the overall file size of the application absolutely minimal. Also, as an SWHX application is essentially a Neko application, it means you have an infinitely extendable framework, which can transfer from machine to machine without requiring a complex installation.

Another benefit of SWHX that is worth bearing in mind is one that has been mentioned consistently throughout this book. Providing you develop your applications with haXe, you will essentially be developing the entire application with a single language.
Installing SWHX

Acquiring the SWHX library is simple. First, open a command console or terminal, and at the command line, enter the following:

```
haxelib install swhx
```

Remember to press Return. This should now download all necessary files from the haXe website. Simple, huh?

If you like, you can run a series of examples by following the installation with the line:

```
haxelib run swhx
```

The source for these examples is available to peruse in the folder `lib/swhx` located in the haXe installation directory. It is strongly advised, once you are finished reading this chapter, to take a look at the examples and get familiar with how they work.

Getting Started with SWHX

Now you’re pumped and ready to go. You already know 99 percent of what is required to produce an SWHX application, if you’ve worked through this book chapter by chapter. All you need to know now is how to get the whole thing started and how to get your Flash and Neko layers talking to one another. You’ll get to the latter a little later in the chapter. For now, here’s a look at how one might actually get a simple SWHX application running.

Following is a simple Neko application utilizing the SWHX library. As the only difference required within a Flash movie is relative to its communicability with the Neko layer, you don’t need a script for that just yet:

```java
class App {
    static function main() {
        swhx.Application.init();
        var window = new swhx.Window( "Sample Application", 200, 200 );
        var server = new neko.net.RemotingServer();
        var flash = new swhx.Flash( window, server );
        flash.setAttribute( "src", "sample.swf" );
        flash.start();
        window.show( true );
        swhx.Application.loop();
        swhx.Application.cleanup();
    }
}
```

To run this code, you can substitute `sample.swf` with the location of any SWF file. Bear in mind, though, that the window that is displayed will show with the dimensions 200 × 200 pixels.
Part III: Extending the Possibilities

So, this is about as basic as it gets. There are a number of extra properties and functions you can use to further customize the capabilities of SWHX, some of which you’ll look at now, some later. To get a real feel for what’s happening, however, you’ll break this example down and see what’s going on.

To begin, you made a call to the static function init of the class swhx.Application. This function can accept two arguments, both optional:

```java
```

The first argument specifies the version of Flash player you want to use, while the second is the preferred path to the Flash player. If either of the arguments is missing, SWHX will default to the latest available Flash player, in a default location. If the player doesn’t appear to exist in the default location, SWHX will attempt to download the latest plug-in from the Adobe website. Failing this, a quick search is then made to see if the plug-in is already available from an installed Mozilla- or Netscape-compatible Internet browser.

The next line of the example creates a new operating system window, with a given title, width, and height:

```java
```

On Windows and Mac OSX, the given window will derive from the native interface API, while on Linux, the window will derive from the GTK library.

Next, an instance of the Neko remoting library is instantiated, along with an instance of a new SWHX Flash object, which is passed both the window data and the remoting library instance. You now have all the necessary objects to run the application; all that’s needed is to finish setting the necessary parameters for your movie and then set SWHX running.

As the Flash player plug-in used by SWHX is that of the Netscape browser family, it has been optimized so that it can work well with string-based XML values. Thus, the attribute functionality for the plug-in is oriented toward key/value pairs, passed as strings. Therefore, when specifying which SWF movie the Flash player plug-in should load, it must be done using the setAttribute function of the plug-in, and should pass both the attribute type, which is src, and the path to the SWF movie. In HTML, this same data would look like this:

```html
<embed src="sample.swf" />
```

You will notice that, as it is a Netscape-compatible plug-in rather than an Internet Explorer-compatible plug-in, the values match those of the HTML embed tag used by Mozilla Firefox et al., rather than that of the param tag.

Once the movie attributes are set, it is then time to start the Flash object, which loads and plays the specified movie, shows the as yet hidden window, and sets SWHX into an event loop. From here on in, any calls made to and from the Flash movie will need to occur within a callback, which will logically have to be made by the Flash movie. Of course, if you wish, you can forgo any Neko communications whatsoever, but you will be missing out on an incredible vista of possibilities if you do.
At any time, you can exit the SWHX loop from a Neko callback using:

```
swhx.Application.exitLoop();
```

When the application eventually ends, for whatever reason that might be, the SWHX loop will break, and the `swhx.Application.cleanup` function will be called, which will enable the garbage collector to do its bit. At this point, you should try to forcibly clean up any stray database connections, abstract pointers, and other such memory hogging connections, so as to play nicely with your operating system.

## Working with Windows

One of the most common reasons for working with an SWF2Exe style framework is to utilize alternative window functionality. The features supplied automatically with the standard Flash projector aren’t exactly extensive, and can often be a little restrictive for even the most simple desktop Flash application. Therefore, to be given the freedom to control a window size and location, specify the window border type and controls, and even make the window completely transparent, is something that will appeal to many.

### Window Decoration Flags

Instigating this level of control is very simple with SWHX, thanks to the **flags** parameter of the Window class constructor. There are numerous combinations, some of which require other flags to be set before they will work. Table 18-1 details the possible window flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WF_FULLSCREEN</td>
<td>Sets the window to full screen. Any window decoration is removed.</td>
</tr>
<tr>
<td>WF_TRANSPARENT</td>
<td>Shows the content of the window, that is, the Flash movie, without showing any window decoration.</td>
</tr>
<tr>
<td>WF_DROPTARGET</td>
<td>Activates the windows drop target feature, whereby any icons dropped onto the window will fire a <code>dropTarget</code> event.</td>
</tr>
<tr>
<td>WF_PLAIN</td>
<td>Produces a window with no decoration. The window is merely a rectangle with a thin border.</td>
</tr>
<tr>
<td>WF_ALWAYS_ONTOP</td>
<td>Forces the window to always appear above all other windows.</td>
</tr>
<tr>
<td>WF_NO_TASKBAR</td>
<td>Hides the windows icon from the taskbar. Useful for dialogs.</td>
</tr>
</tbody>
</table>

The flags can be OR’d together in order to combine several effects at once. For example, if you wish to have a transparent window that is always on top and doesn’t show in the taskbar, you can use the following:

```
var window = new swhx.Window( "Sample Application", 200, 200,
    swhx.Window.WF_TRANSPARENT | swhx.Window.WF_ALWAYS_ONTOP | swhx.Window.WF_NO_TASKBAR );
```
Part III: Extending the Possibilities

Window Properties

So, the flags help with the initial construction of a window, but numerous other window-based capabilities can be switched on or off, or require some configuration in order to work. These can be accessed via the Window class’s various properties.

Some of the more important Window properties, or at least those that tend to get used more often, are those that affect the window dimensions and location: width, height, top, and left. Only recently, these four properties were used to a great extent when building a desktop buddy. This is a character that aimlessly walks around your desktop providing the occasional cute interaction. The point is, just because it’s not commonly done, much of a window’s properties can be accessed and modified at run time, providing extensive capabilities for applications development.

When developing dynamic transparent windows, it is worth noting that SWHX does not yet support a regional rendering engine; thus full-screen windows with small animations that wander the desktop still require that the full window be rendered frame by frame. This can be expensive in terms of memory. Therefore, where possible, aim to use a moving, resizing window that will ultimately use fewer resources.

For a full lowdown of the properties supported by the Window class, see Table 18-2.

Table 18-2

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resizable : Bool</td>
<td>Determines whether the window can be resized.</td>
</tr>
<tr>
<td>maximizeIcon : Bool</td>
<td>Determines whether the maximize icon is enabled/shown.</td>
</tr>
<tr>
<td>minimizeIcon : Bool</td>
<td>Determines whether the minimize icon is enabled/shown.</td>
</tr>
<tr>
<td>width : Int</td>
<td>Determines the windows width.</td>
</tr>
<tr>
<td>height : Int</td>
<td>Determines the windows height.</td>
</tr>
<tr>
<td>left : Int</td>
<td>Determines the windows x coordinate location on the screen.</td>
</tr>
<tr>
<td>top : Int</td>
<td>Determines the windows y coordinate location on the screen.</td>
</tr>
<tr>
<td>handle : Void (abstract)</td>
<td>A C level handle of the window, for use in extending in Neko ndll modules. It is a read-only variable.</td>
</tr>
<tr>
<td>fullscreen : Bool</td>
<td>Determines whether the window is set to fill the whole screen.</td>
</tr>
<tr>
<td>transparent : Bool</td>
<td>Determines whether the window is set to transparent (read-only).</td>
</tr>
<tr>
<td>flashRunning : Bool</td>
<td>Determines whether the contained flash movie is running (read-only).</td>
</tr>
<tr>
<td>dropTarget : Bool</td>
<td>Determines whether the window can accept drag-and-drop files.</td>
</tr>
<tr>
<td>title : String</td>
<td>Sets and returns the windows title.</td>
</tr>
<tr>
<td>plain : Bool</td>
<td>Determines whether the window was created with the WF_PLAIN flag set.</td>
</tr>
<tr>
<td>minimized : Bool</td>
<td>Determines whether the window is minimized.</td>
</tr>
<tr>
<td>maximized : Bool</td>
<td>Determines whether the window is maximized.</td>
</tr>
</tbody>
</table>
Window Events

Much as you would expect when working with native operating system windows, much of what you wish to achieve with regard to window interaction is accomplished through the use of events. The Window class exposes several functions, which you can override in order to hook into these events, and apply your own code in haXe for supplying any necessary functionality that should be run when such events are fired.

Drag Event

One of the coolest events supported by SWHX is the drag event. The theory is that any Flash MovieClip in your running movie can act as a handle for dragging the container window around the screen. This becomes especially important when dealing with windows that have the WF_TRANSPARENT flag set at creation time, as without any window decoration, you are left with no handles with which to move your window.

A drag event is fired by calling the Window.drag function. This is normally be called direct from a Flash movie, though under the right circumstances, it can also be called from the Neko layer. This event persists until the pressed mouse button is released.

onClose Event

In all likelihood, when you create an SWHX application, the initialization and display of a window will be the first thing you do. Likewise, when the window is closed, you will probably want your application to end. The problem arises that you will not be able to plan for a window to be closed, so your application could be in any state when such an event occurs. For example, it might be that at the time a window closes, the Neko layer has an open connection to a database. Simply disposing of an application without shutting down a database connection properly would be wasteful of machine resources, and will be the cause of a memory leak. At worse, depending on how the particular database handles impolite connection severances, you might just lose some important data.

Luckily, SWHX provides the useful onClose method, which is called the moment a window handled by the SWHX layer is closed. By overriding this method, one could easily ensure that all important connections to external resources are safely closed, and all allocated memory is released. The onClose method expects a return value of type Bool; by returning true, the window is safe to close, while false forces the window to remain open.

swhx.Application.loop is exited the moment all windows are closed, allowing any code that follows the loop to be run. When building an application utilizing only one window, it pays to be safe with regard to how the application cleans up after itself by putting all cleanup code inside the onClose method.

onMinimize and onMaximize Events

The onMinimize and onMaximize methods are, surprisingly, called whenever the window is either minimized or maximized. These event handlers can be useful if you wish to halt a part of the application if the window is minimized, and to restart when maximized.
Part III: Extending the Possibilities

onRightClick Event

The onRightClick method is fired when the right mouse button is pressed over the windows client area. This method is especially useful should you wish to display a context menu of some sort, or to enable some form of alternative Flash-based interaction. Like onClose, onRightClick expects a Boolean return value; returning false will stop the right-click event bubbling to the Flash player, thus quashing the default Flash player context menu, while true allows the event bubbling.

onFilesDropped Event

The onFilesDropped method is called only if and when the dropTarget property of the window instance is set to true. This particular event method is passed an Array value of type String, which lists the paths and filenames of each file’s icon to have been dropped onto the windows client area. This proves useful when you want to create applications that respond to external files rather than, say, having to specifically select a load option:

```javascript
window.dropTarget = true;
window.onFilesDropped = function( files : Array<String> )
{
    if ( files.length > 0 )
    {
        for ( i in files )
        {
            neko.Lib.print( i );
        }
    }
    return true;
}
```

Custom Events with Message Hooks

The aforementioned events cover a number of the native event types supported by the window as handled by the operating system. However, any veteran C/C++ coder will know that an operating system’s event queue supplies a staggering number of event types that can normally be caught and dealt with should the need arise. While not being able to supply handlers to absolutely every single event type available, the SWHX library does provide a means to trap specific events through a proxy, so that code supplied in the Neko layer can be called when such an event is fired. What’s more, the numeric message parameter variables existent in such event queue handler methods can be acquired in the Neko layer for analysis. This proves invaluable in situations such as handling the results of a menu selection, for example.

When creating a hook to a specific event type, one must know the actual numeric constant value associated with the event in the operating system–specific C or C++ libraries. This information can be found by browsing the library header files or by performing a quick search in google for C-based event constants that are prepended with the characters win_. Once this information is known, you can then proceed to acquire a hook into the events messaging loop using the addMessageHook method of the Window class:

```javascript
var hook : swhx.MessageHook = window.addMessageHook( intEventID );
```
Chapter 18: Desktop Flash with SWHX

The returned hook value can then be used to assign a Neko function for handling the event:

```javascript
hook.setNekoCallback( evtCallback );
```

The hook value can also be used within the callback method to acquire one or both of the event parameters, which are assigned to the MessageHook instance as the parameters p1 and p2:

```javascript
var evtCallback = function()
{
    neko.Lib.print( hook.p1 + ", " + hook.p2 );
}
```

It is worth noting at this point, although we are not sure how legitimately, that while a reference to a static function can be passed to `MessageHook.setNekoCallback`, we often found that when an application has been running for some time, the reference to a static function will tend to get lost, causing a nasty crash. We have yet to experience this same bug when using a local function.

As an alternative to assigning a Neko function reference as a callback to a window event, one can also directly assign a C-level function pointer, available within Neko as a Void abstract value, directly to the event messaging loop. This is accomplished using the `setCCallback` of the `MessageHook` instance:

```javascript
hook.setCCallback( myCLevelFunction );
```

The passed C function pointer must assume the following specific signature:

```c
void *(*callback) ( callbackData *void, void *id1, void *id2, void *p1, void *p2 );
```

Because it is beyond the scope of this book, the parameters are not detailed here, though you can find the information listed within the SWHX ndll source code.

Communication Between Flash and Neko

So far, you have looked at how an application might display a Flash movie, with some highlights into message handling from the perspective of the window. However, you have not yet looked into the important aspect of handling communications between Flash and the window, or Flash and the Neko layer.

For those of you who have used other SWF2Exe frameworks to mobilize your Flash applications, you’re in for a treat. Usually, when developing against SWF2Exe frameworks, one needs to have to hand a large document detailing all the available API functionality for inclusion into the Flash file. This can become tedious as, not only does it make the framework limited in terms of capability and provide yet more code for the developer to learn, but it also means the Flash developer has to hard code functionality into the presentation layer, that would be better off elsewhere.
Part III: Extending the Possibilities

**Asynchronous Communication**

The simplest way to communicate between Flash and Neko is to handle calls from an FSCommand in the Neko layer. The Flash object you create in the Neko layer, which eventually gets hosted by a Window instance, provides a method, which you can override called `onFSCommand`:

```javascript
flashObject.onFSCommand( command : String, parameters : String ) : String
```

The idea of this function is that, should you wish to pass a command to the Neko layer, you can do so in the most generic of terms, while not breaking existing ActionScript code. Unfortunately, the FSCommand function can only send two parameters, both of which are of type String. However, as long as you delimit multiple values sent as a parameter, you can quite easily reconstruct the data in the Neko layer. Another drawback of this method is that, being asynchronous, the communication is limited to Flash to Neko calls, incapable of even return values.

Anyway, for some purposes, this can be a great option. Take a look at how this can be put to use by creating an app whereby the Flash layer performs a timer, and the Neko layer acts accordingly. First, the Flash code:

```javascript
import flash.Lib;
class UI extends flash.MovieClip {
  static var className : String = "UI";
  static var register : Dynamic = flash.Lib.registerClass( className, UI );
  static var count : Array<Int> = [0,0];
  static var scale : Array<Int> = [0,0];
  static var dir : Array<Bool>
  static function main()
  {
    Lib.current.attachMovie( UI.className, "ui", 0 );
  }
  public function new()
  {
    drawRect( this, 20, 20, 0xFF0000 );
    this.onEnterFrame = function()
    {
      if ( dir[0] == true )
      {
        if ( ++scale[0] >= 10 ) dir[0] = false;
      }
      else
      {
        if ( --scale[0] <= -10 ) dir[0] = true;
      }
      count[0] = 100 + scale[0];
      if ( dir[1] == true )
      {
        if ( ++scale[1] >= 20 ) dir[1] = false;
      }
      else
      {
        if ( --scale[1] <= -20 ) dir[1] = true;
      }
    }
  }
}
```

count[1] = 100 + scale[1];
Lib.fscommand( "moveUpdown", Std.string( count[1] ) );
Lib.fscommand( "moveLeftRight", Std.string( count[0] ) );
}
}

public static function drawRect( m : flash.MovieClip, w : Int, h : Int, c : Int )
{
    m.beginFill( c );
m.moveTo( 0, 0 );
m.lineTo( w, 0 );
m.lineTo( w, h );
m.lineTo( 0, h );
m.lineTo( 0, 0 );
m.endFill();
}

Compile this for Flash as ui.swf. Next, create a new document and enter the following:

class NekoLayer
{
    public static function main()
    {
        var inst = new NekoLayer();
    }
    public function new()
    {
        swhx.Application.init();
        var window = new swhx.Window(
            "Test Application",
            200,
            200,
            swhx.Window.WF_TRANSPARENT );
        var flash = new swhx.Flash( window, null );
        flash.setAttribute( "src", "ui.swf" );
        flash.onFSCommand = function( p1 : String, p2 : String )
        {
            switch ( p1 )
            {
                case "moveLeftRight":
                    window.left = Std.parseInt( p2 );
                case "moveUpdown":
                    window.top = Std.parseInt( p2 );
            }
            return null;
        }
        flash.start();
        window.show( true );
        swhx.Application.loop();
        swhx.Application.cleanup();
    }
}

Compile this code for Neko as app.n.
Part III: Extending the Possibilities

If you now run the Neko application, you should see a small red square perform a figure-eight dance in the top left corner of your monitor. Neat, huh? All that’s really happening is that a sequence of values alternate from −20 to 20 for the vertical position and from −10 to 10 for the horizontal. The values are then consistently passed to Neko through the FSCCommand calls, and the window position is updated accordingly.

Normally, one could perceive several ways in which to optimize this code. For one, both horizontal and vertical values could be passed together, requiring fewer calls from one layer to the other. Optimizing further still, one could remove the need to move the whole window, and instead perform the same animation feat within Flash itself. This would mean that the window itself doesn’t move, but the effect is the same. The point is that, you need to make sure that the calls you make to the Neko layer are required. Flash to Neko communication isn’t exactly expensive, but it can add more cost than is necessary and it does tend to make code seem more complex.

Synchronous Communication in haXe

SWHX provides a synchronous communication system, thanks to the already existent haXe Remoting API. As you will already know from Chapter 15, “Putting It All Together with haXe Remoting,” when developing web applications using Neko and Flash, both layers can transfer data to each other seamlessly without the need to convert data types or manually serialize data. SWHX for desktop applications can use the same remoting infrastructure, combined with a few simple proxy functions that make calling from one layer to the next incredibly simple indeed. This leaves the developer with the job of creating solid functionality, not the difficult job of creating clean code. Well, at least, not more than usual.

Here’s a look at an example. Create a new document and enter the following:

```haxe
class NekoLayer
{
    public static function main()
    {
        var inst = new NekoLayer();
    }
    public function new()
    {
        swhx.Application.init();
        var window = new swhx.Window( "Test Application", 400, 300 );
        var server = new neko.net.RemotingServer();
        server.addObject( "App", NekoLayer );
        var flash = new swhx.Flash( window, server );
        flash.setAttribute( "src", "ui.swf" );
        flash.start();
        window.onRightClick = function()
        {
            var cnx = swhx.Connection.flashConnect( flash );
            var d = readData( "test.txt" );
            cnx.UI.showData.call( [ d ] );
            return false;
        }
    }
}```
window.show( true );
swhx.Application.loop();
swhx.Application.cleanup();
}

public static function writeData( file : String, content : String )
{
    var f = neko.io.File.write( file, true );
    f.write( content );
    f.close();
}

public static function readData( file : String )
{
    var f = neko.io.File.read( file, false );
    var d = f.readAll();
    f.close();
    return d;
}  

Now, compile this as a Neko application file called app.n, then create another empty document and enter the following:

import flash.Lib;
class UI extends flash.MovieClip
{
    static var className : String = "UI";
    static var register : Dynamic = flash.Lib.registerClass( className, UI );
    static var receive_tf : flash.TextField;
    static var send_tf : flash.TextField;
    static var send_mc : flash.MovieClip;
    static function main()
    {
        Lib.current.attachMovie( UI.className, "ui", 0 );
    }
    public function new()
    {
        send_tf = this.createTextField( "send_tf", 1, 10, 10, 150, 20 );
        send_tf.border = true;
        send_tf.type = "input";
        receive_tf = this.createTextField( "receive_tf", 2, 10, 40, 150, 100 );
        receive_tf.border = true;
        send_mc = this.createEmptyMovieClip( "send_mc", 3 );
        send_mc._x = 170;
        send_mc._y = 10;
        drawRect( send_mc, 20, 20, 0x00FF00 );
        send_mc.onRelease = function()
        {
            var cnx = swhx.Connection.desktopConnect();
            cnx.App.writeData.call( [ "test.txt", send_tf.text ] );
            send_tf.text = "The test.txt file has been created!";
        }
    }
}
Part III: Extending the Possibilities

(continued)

```java
public static function showData( val : String )
{
    receive_tf.text = val;
}
public static function drawRect( m : flash.MovieClip, w : Int, h : Int, c : Int )
{
    m.beginFill( c );
    m.moveTo( 0, 0 );
    m.lineTo( w, 0 );
    m.lineTo( w, h );
    m.lineTo( 0, h );
    m.lineTo( 0, 0 );
    m.endFill();
}
```

Compile this class into a Flash file called `ui.swf`.

Now, if you run `app.n` from the command line, you should be presented with a window displaying two text fields and a square green button. By entering text into the top text field, then pressing the green button, you cause the Flash layer to send the content of the text field to the Neko layer, which then gets written to a file `test.txt`. If you then right-click the Flash movie, you invoke the `onRightClick` event, which forces the Neko layer to read the content of the text file into a variable, and send it to the `showData` function in the Flash layer, which writes it to the second text field.

So, how does this work? You’ll notice, primarily, a couple of lines of code within each class that handles the sending of data from one layer to the other. In the Flash layer, it reads:

```java
var cnx = swhx.Connection.desktopConnect();
cnx.App.writeData.call( [ "test.txt", send_tf.text ] );
```

Here, a connection object is created from the command `swhx.Connection.desktopConnect`. The connection object is then used to proxy the call to the `writeData` method of the `NekoLayer` class, sending it the string `test.txt` and the content of the `send_tf` text field.

So, then, how does the call know what method from what class it is calling? Well, let’s examine the call proxy code in detail:

```java
cnx.App.writeData.call( [ "test.txt", send_tf.text ] );
```

This line can be analyzed as:

```java
[connection obj].[class].[method].call( parametersArray );
```

If you look back closely at the `NekoLayer` class, there is a line of code that says:

```java
server.addObject( "App", NekoLayer );
```

This line registers the `NekoLayer` class with the remoting layer as `App`. Therefore, when you call the class from the Flash layer, reference the class as `App`, even though its real physical name is `NekoLayer`. 
Chapter 18: Desktop Flash with SWHX

You can reference other classes like this, if you like, by making subsequent calls to the addObject method, so that each of the classes is available to the connection object in the Flash layer. If you don’t reference a class with this method call, you won’t be able to access its methods from Flash.

Now, what about the calls from Neko to Flash? Well, in the Neko code, two lines are similar to the connection code specified in the Flash layer, which look like this:

```javascript
var cnx = swhx.Connection.flashConnect( flash );
cnx.UI.showData.call( [ d ] );
```

Here, the connection object already knows about the class it is calling from the passed Flash object parameter to the flashConnect method. This means that, while you must register any Neko layer classes that will be called from Flash, you do not need to register the Flash classes that you will call from Neko. However, one important point to remember is that, when specifying the class you wish to call within the Flash movie, you must specify its package path as well as its class name. So, if the class UI belonged to the package com.flash.graphics, then the proxy call would look like this:

```javascript
cnx.com.flash.graphics.UI.showData.call( [ d ] );
```

One final point to discuss about SWHX communication is with regard to return values. Now, this might seem common sense, but you must try to refrain from making a call from one layer to the next within a function that itself will be called from the partnering layer. This shouldn’t cause any issues itself in relation to resource management or application stability, but doing so is very bad from a programming code of practice perspective, and could result in a perpetual loop that might be very hard to locate, should numerous functions make calls to each other.

**Synchronous Communication in ActionScript**

The SWHX library provides extensions for both ActionScript 2 and ActionScript 3, so you don’t have to use haXe to compile your SWF’s in order to make use of SWHX. Using the extensions is simple, requiring a call to initialize the extension, followed by the necessary calls to the Neko layer when you want to execute Neko-based functions.

To demonstrate, here is an ActionScript 2 version of the previous example:

```actionscript
import swhx.Api;
class UI extends MovieClip
{
    static var className : String = "UI";
    static var register : Object = registerClass( className, UI );
    static var receive_tf : TextField;
    static var send_tf : TextField;
    static var send_mc : MovieClip;
    static function main()
    {
        _root.attachMovie( UI.className, "ui", 0 );
    }
}
```

(continued)
public function UI()
{
    Api.init( this );
    send_tf = this.createTextField( "send_tf", 1, 10, 10, 150, 20 );
    send_tf.border = true;
    send_tf.type = "input";
    receive_tf = this.createTextField( "receive_tf", 2, 10, 40, 150, 100 );
    receive_tf.border = true;
    send_mc = this.createEmptyMovieClip( "send_mc", 3 );
    send_mc._x = 170;
    send_mc._y = 10;
    drawRect( send_mc, 20, 20, 0x00FF00 );
    send_mc.onRelease = function()
    {
        swhx.Api.call( "App.writeData", "test.txt", UI.send_tf.text );
        UI.send_tf.text = "The test.txt file has been created!";
    }
}

public static function showData( val : String )
{
    receive_tf.text = val;
}

public static function drawRect( m : MovieClip, w, h, c : Number )
{
    m.beginFill( c );
    m.moveTo( 0, 0 );
    m.lineTo( w, 0 );
    m.lineTo( w, h );
    m.lineTo( 0, h );
    m.lineTo( 0, 0 );
    m.endFill();
}

This should compile with no problems for Flash versions 6 to 8, provided you set a class path to the SWHX ActionScript 2 extension files.

As you can see from the aforementioned class, little was changed from the haXe version. Obviously, there were necessary syntax changes required from converting the script from one language to the other, but with regard to converting the class for use with the new SWHX ActionScript 2 classes, all you were required to do was to initialize the API with a call to swhx.Api.init, passing the current object as its parameter, and to substitute all calls to the Neko layer with a call to swhx.Api.call.

As far as changes go, this was the biggest change. While the haXe version of the SWHX libraries allow the specified Neko class name and method to be specified within the actual dot notation, the ActionScript 2 version requires that the object and method names be specified as a string and passed as the first parameter to the call method.

You’ll also notice that the call has done away with the passing of a single array containing the parameter list, and instead relies upon a variable number of parameters passed in the standard way for functions. Keep this in mind should you become too familiar with the haXe variety of synchronous communication, lest you lose hair trying to work out where your code is going wrong.
Chapter 18: Desktop Flash with SWHX

Using SysTools

While the SWHX library was being developed, its author, Edwin Van Rijkom, decided it best to segregate a chunk of the functionality originally intended for SWHX into a separate library. So was born the SysTools library.

SysTools contains a random assortment of goodies that can prove very useful for desktop applications development, particularly when using SWHX where native operating system features are a little further divorced. For the most part, the features provided by SysTools can vary considerably based on the operating system you are developing for. The features that currently exist have primarily been constructed by Edwin Van Rijkom, Nicolas Cannasse, Ian Thomas, and Lee McColl-Sylvester, but have come about through need. As one developer requires a feature that is not yet present in SysTools, that developer seeks to include it for a current project in the hopes that some other developer with more time will see fit to expand those features for each of the other operating systems.

Table 18-3 provides an outline of the features supported by SysTools, and the operating systems that currently support them.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Box Dialog</td>
<td>All</td>
</tr>
<tr>
<td>Confirmation Dialog</td>
<td>All</td>
</tr>
<tr>
<td>Save File Dialog</td>
<td>All</td>
</tr>
<tr>
<td>Open File Dialog</td>
<td>All</td>
</tr>
<tr>
<td>Folder Dialog</td>
<td>All</td>
</tr>
<tr>
<td>Browser launch with specified URL</td>
<td>Mac and Windows</td>
</tr>
<tr>
<td>Copy and retrieve from clipboard</td>
<td>All</td>
</tr>
<tr>
<td>Retrieve temp directory location</td>
<td>All</td>
</tr>
<tr>
<td>Get pressed keys on keyboard</td>
<td>Mac and Windows</td>
</tr>
<tr>
<td>Write to and from the registry</td>
<td>All</td>
</tr>
<tr>
<td>Retrieve details of an SWF file, such as frame rate and dimensions</td>
<td>All</td>
</tr>
<tr>
<td>System tray icon support</td>
<td>Windows</td>
</tr>
<tr>
<td>Native context, tray, and menu bar menu’s</td>
<td>Windows</td>
</tr>
<tr>
<td>Replace executable application icon</td>
<td>Windows</td>
</tr>
<tr>
<td>Change display resolution</td>
<td>Windows</td>
</tr>
</tbody>
</table>

Not all of these features are covered in this chapter, but you’ll certainly take a look at some of the more intriguing ones.
Part III: Extending the Possibilities

**Creating a System Tray Icon**

When you develop SWHX applications, particularly those utilizing the WF_NO_TASKBAR flag, it can often pay to assign access to the application through an icon in the Windows system tray. This can be particularly impressive should your application behave as a useful tool that might sit running on a user's desktop from the moment it boots up to the moment it shuts down.

Creating a tray icon is simple, providing you already have an icon you would like to use. Icon files have an .ico extension, and are essentially graphic files. If you don't have one lying around that will suit, there are many on the Internet that you can download for free. Alternatively, you could try one of the many shareware icon design applications on the Internet. Just make sure that any icon you use is set to display at 16 × 16 pixels.

Once you have your icon, place it in your project folder and note where it is in relation to the executable that will boot your application. Next, in the part of the code that initializes your application, enter the following line:

```java
tray = new systools.win.Tray( window, iconPath, toolTip );
```

The variable holding the returned reference should be a static variable. This is essential, as you don't want to lose the reference before your application exits. Doing so could lead to the tray icon disappearing prematurely.

The parameters required by systools.win.Tray are, in order, the object instance of the Window class for your primary SWHX window, a string depicting the relative location to the icon file, and a string containing the text to use as a rollover tooltip. You can pass null as the third parameter if you do not want to display any tooltip text. Bear in mind, though, that the tooltip can currently only be set when you initially make a call to create a tray icon.

**Creating Menus**

In our opinion, nothing makes a desktop multimedia application look more professional than native menus, and SWHX supports it all, at least for Windows. Creating menus in SWHX is fairly simple, but as always, here's a look at an example first. This particular example is based on a sample that comes with SWHX:

```java
import systools.win.Tray;
import systools.win.Menus;
import systools.win.Events;
class App
{
    static var flash : swhx.Flash;
    static var window: swhx.Window;
    static var hook: swhx.MessageHook;
    static var thook: swhx.MessageHook;
    static var mhook: swhx.MessageHook;
    static var tray: Tray;
    static var m: Menus;
    static var mm: Menus;
    static var ms1: Menus;
    static var ms11: Menus;
    static var ms2: Menus;
```
static var ms3: Menus;
static function main()
{
    swhx.Application.init();
    window = new swhx.Window( "SWHX Menus", 400, 300 );
    window.onRightClick = function() {
        return false;
    }
    window.onClose = cleanUp;
    var server = new neko.net.RemotingServer();
    server.addObject("App",App);
    flash = new swhx.Flash(window,server);
    flash.setAttribute("src","ui.swf");
    flash.onSourceLoaded = onSourceLoaded;
    flash.start();
    swhx.Application.loop();
    swhx.Application.cleanup();
}
static function onSourceLoaded()
{
    window.show(true);
    mm = new systools.win.Menus( false );
    m1 = new systools.win.Menus( true );
    m1.addItem( "sub 1-1", 4 );
    m1.addItem( "sub 1-2", 5 );
    m1.addItem( "sub 1-3", 6 );
    m1l = new systools.win.Menus( true );
    m1l.addItem( "sub 1-4-1", 4 );
    m1l.addItem( "sub 1-4-2", 5 );
    m1l.addItem( "sub 1-4-3", 6 );
    m1.addSubmenu( m1l, "sub 1-4", 13 );
    m2 = new systools.win.Menus( true );
    m2.addItem( "sub 2-1", 7 );
    m2.addItem( "sub 2-2", 8 );
    m2.addItem( "sub 2-3", 9 );
    m3 = new systools.win.Menus( true );
    m3.addItem( "sub 3-1", 10 );
    m3.addItem( "sub 3-2", 11 );
    m3.addItem( "sub 3-3", 12 );
    mm.addSubmenu( m1, "option 1", 1 );
    mm.addSubmenu( m2, "option 2", 2 );
    mm.addSubmenu( m3, "option 3", 3 );
    m = new systools.win.Menus( true );
    m.addItem( "option 1", 1 );
    m.addItem( "option 2", 2 );
    m.addItem( "option 3", 3 );
    hook = window.addMessageHook( untyped Events.RBUTTONUP);
    hook.setNekoCallback( mouseRButtonHook );
    mhook = window.addMessageHook( untyped Events.MENUEVENT);
    mhook.setNekoCallback( menuClickHook );
    thook = window.addMessageHook( untyped Events.TRAYEVENT);
    thook.setNekoCallback( TrayClickHook );
    tray = new Tray( window, "swhx_16x16.ico","Hello Tray!" );
}
Part III: Extending the Possibilities

(continued)

```javascript
static function mouseRButtonHook ()
{
    trace( "Option " + m.showPopup( window.handle ) + " was selected!" );
    return 0;
}
static function menuClickHook()
{
    trace( "Option p1: " + mhook.p1 );
    trace( "Option p2: " + mhook.p2 );
    trace( "Callback: " + hook.callbackData );
    return 0;
}
static function TrayClickHook()
{
    if ( Std.string(thook.p2) == Std.string(Events.RBUTTONUP) )
    {
        trace( "Option " + m.showPopup( window.handle ) + " was selected!" );
        return 0;
    }
}
static function cleanUp()
{
    window.removeMessageHook(hook);
    window.removeMessageHook(mhook);
    window.removeMessageHook(thook);
    return true;
}
```

If you compile and run this code, you should find a nice menu at the top of the window, as well as right-click context menus assigned to the window and to the tray icon in the system tray.

As a word of warning, we’ve set the callback functions as static methods of the class. However, we have had issues with class methods as callback handlers for anything linked to SWHX message hooks, as they tend to get lost. Therefore, when creating callbacks for SWHX message hooks, assign local defined functions, instead.

So, how does this work? Well, first, take a look at the creation of the window message hooks. They were mentioned earlier in this chapter, and are an easy way to assign native links to the windows within which the Flash movies are located. It is here that the magic occurs. As you can see from the preceding class, callback handlers were specified for the events pertaining to RBUTTONDOWN, MENUEVENT, and TRAYEVENT. These are merely integer values that match those specified by the Win32 API. You don’t have to use these values. If you like, you could instead opt for a menu to appear when a user double-clicks with the left mouse button upon the window.

Anyway, once the callbacks are called, the showPopup method of the defined menu objects are called, which in turn display the menu to the user. showPopup is blocking, which means that the Neko script will halt until an option is selected by the user. If an option is selected, then an integer value representing that option is returned; otherwise, should the user click elsewhere outside of the visible menu, then zero is returned. You can make a call to showPopup without waiting for a message to be sent by Windows. Simply make the call to showPopup and the associated menu will appear wherever the mouse cursor is located. This is great for assigning native menus to individual elements of a running Flash movie.
showPopup isn’t necessary for menus assigned to the top of the window, as it handles itself, until an option is selected, at which point its reference value is returned to the Neko callback.

So, what about creating the menu objects? Well, as with the tray icons, you must first create a static variable container for the menu. Once this is done, you must initialize the variable by passing it a new instance of the Menus class:

```javascript
var menu : Menus = new systools.win.Menus( isPopup );
```

The call to the Menus constructor requires a Boolean value that is true if the menu will be a pop-up, context style menu, or false if it is to be attached to a menu bar.

Once you have your menu object, you can then start adding items. Items are either selectable entries, dividers, or submenus. To add a selectable item, you use addItem:

```javascript
menu.addItem( text, id );
```

The first parameter to addItem is the text that will display in the menu for the item, while the second parameter references that integer value that will be returned should it be selected. This integer value also represents the location of the item, as all items are shown in numerical order, with the lowest value integer sitting at the top of the menu and the highest at the bottom. You must make sure that the passed id is unique.

Occasionally, you will want to break up your menu items with the use of dividers. These are easy to implement. You simply use the addDivider method, passing it the unique location of the divider, like this:

```javascript
menu.addDivider( id );
```

Finally, you move on to submenus. Essentially, a submenu is an item just like any other entry, with the exception that submenus are menus in their own right. There’s no reason why you can’t show a menu on its own one minute, and then as a submenu for a parent menu the next.

To add a submenu, you use the addSubmenu method:

```javascript
menu.addSubmenu( menu2, “my submenu”, id );
```

Here, the first parameter is an instance of the Menus class, the second is the caption shown in the parent menu, which represents the submenu, and the third parameter is the id.

### Using Dialogs

SysTools provides a number of dialogs for use in your developments that are simple to use, yet extremely useful. Merely being able to alert a user to a situation in one line of code, without resorting to designing a message box and api in Flash, will save hours of coding time. What’s more, you’ll not only be able to provide information, but collect information, too.
Part III: Extending the Possibilities

The Message Box
The message box is one of the most frequently used of all dialog boxes. It is a simple, single button dialog that allows for the straightforward delivery of important information to the user.

Creating a message box couldn’t be easier. All that is required is a single line of code:

```javascript
systools.Dialogs.message( title, message, isError );
```

The first parameter to this function call represents the title string at the top of the dialog pop-up, the second parameter is the text message that is displayed on the pop-up, and the third is an optional Boolean value that, if true, displays an error icon; otherwise it is an informational icon.

The Confirm Message Box
The confirm message box provides a means for a user to select from a choice of options, yes or no. Its primary use is to allow the user of the application to respond to important actions, such as whether to save a document when shutting down the application. This feat is provided by offering two buttons, rather than the normal one. If the user clicks on the first button, labeled yes, then the function that derives the confirm message box returns `true`; otherwise `false` is returned.

The confirm message box has the same signature as the standard message box, with the exception that it returns a Boolean value:

```javascript
systools.Dialogs.confirm( title, message, isError );
```

Summary
This chapter covered quite a bit. SWHX is an excellent library for haXe that makes both haXe and Neko valid contenders in the desktop applications industry all by itself. And while the possibilities for this library are far reaching, it still remains simple enough to use that it can be covered in a single chapter.

In this chapter, you learned:

- Why SWHX is the best option for SWF2Exe projects everywhere
- How to install SWHX
- How to initialize and utilize SWHX
- How to manage communication between Flash and Neko.
- How to use dialogs with SysTools
- How to use menus and system tray icons

In the next chapter, you look at utilizing threads for multithreaded applications development, and sockets for multiuser applications development. You also take a look at the `ThreadedServer` class, which utilizes both threads and sockets for optimized server-based multiuser applications.
The haXe language provides a flexible platform for the development of multimedia applications with Flash, but also, thanks to the sheer power of the Neko Virtual Machine, you can now build executable desktop multimedia applications, without a single Flash file in sight. This is thanks to two very powerful, cross-platform libraries called the Simple DirectMedia Library (SDL) and Allegro.

Both of these libraries have been bound to the Neko Virtual Machine as ndll modules. The Allegro library has been included by the nGame module, which provides a simple haXe bindings layer into the Allegro API. The SDL library, however, forms the foundation of the Neko Media Engine (NME) framework, which also provides functionality for features not directly supported by the SDL library, such as a circular collision detection algorithm and a tile map manager. For this reason, this chapter focuses more closely on the use of the NME module, in particular with regard to the development of games.

Although this chapter is titled “Multimedia with Neko,” it actually focuses more upon the development of games. The reason for this is that games use almost all of the functionality that many multimedia applications require, and utilize these at much greater speeds than most multimedia applications. On the other hand, multimedia applications and games are very similar, and one could often argue that what little difference exists between them is merely a matter of perception.

In short, this chapter will cover:

- How to use Neko for games development
- Which library you should use: NME versus nGame
- How to create a simple game loop
- How to use timers
- How to create and animate Sprite objects
- How to deal with player interaction
Why Use Neko for Games Development?

When developing games, or indeed, high-performance multimedia applications, it is often necessary to utilize a fast, capable language such as C or Pascal. The Neko run time, having been efficiently written in C, provides a platform that should be quite suitable for games development. Although we’re pretty sure this hasn’t been done with regard to commercial games, just yet, commercial games do utilize scripting languages all the time. In fact, you may be surprised to know that the Eve Online game, which is a massively multiplayer online (MMO) product, claims the right to the most consecutive players online at once on a single server and uses Python for most of its server-side functionality. While we haven’t seen evidence toward comparable benchmarking between Neko and Python, through our own experiences, we have found Neko to be particularly superior.

So what does this mean from a development perspective? Well, with such a fast runtime environment, one could decide to use the Neko Virtual Machine to power the logic behind an entire game client. While this may prove to be a little slower than the equivalent written in C, the effects of such an effort would not likely suffer a great deal, and would certainly benefit astronomically by being able to leverage the wonderful haXe language.

When one contemplates the components that make up a modern game, it can appear daunting that, not only must graphics, sound and interactivity be taken care of, but also communications, data storage and various other tidbits. As you’ve seen earlier in the book, communications is one of the haXe language’s strongest features, while the SPOD layer takes care of any data storage issues. All you’re left to work out is which graphics library you’d like to bind to a Neko module.

Another boon to the whole games development with Neko proposal is that one could quite possibly press the cross-platform development notion, making a game that would function on numerous operating systems. Indeed, if you were to utilize the power of the OpenGL library, you would have a wealth of power at your fingertips with minimal restriction to platform.

Remember, many a successful game has been written without the leverage of 3D graphics capabilities. This chapter demonstrates what is possible with Neko and a cross-platform 2D library, which itself has been used in several recent multimillion dollar games.

Neko’s NME Is Your Friend

The NME library is the result of around two months labor, plus about ten years or so of labor that went into the development of the SDL library. As such, it’s a little early to discuss how great the NME library is, but certainly has enough backup to take a closer look.

Many great game development engines are available. You need only perform a quick search in Google to uncover a treasure trove of tools capable of rendering a playable game. However, of all these libraries, the two that stand high above all others are the Allegro library and the SDL library. When the time came that a decent 2D games library was needed for Neko, a choice had to be made between these powerful competitors, and failing in the task of making a decision, we wrapped both. While the debate of which library is the better remains within the eye of the beholder, we chose to concentrate most of the efforts into the SDL library, as its share of features are quite a bit more open to extending and improving, even if those features are not as initially extensive as the Allegro offering.
Chapter 19: Multimedia with Neko

The power of the SDL library is not something that should be overlooked. Granted, the capabilities available within the code of the library are oriented toward the development of 2D applications, but it also provides a stable foundation for the development of OpenGL applications, and, thus, 3D application development. The SDL library often proves one of the better frameworks for developing with OpenGL, which is why well-known commercial games, such as Unreal Tournament 2004, are using it.

The NME library utilizes SDL, but then to increase its capabilities, it also provides a number of classes that facilitate extra functionality, such as collision detection algorithms and animation. Over the course of NME’s lifetime, it is expected that new features, such as tile mapping support and OpenGL capabilities, will also make their appearance.

Getting Started with NME

Let’s get down to business. While the topic of games and multimedia development can span volumes, cramming all the necessary information on how to develop with NME will simply have to fit into this solitary chapter. For that reason, you’ll start with a hefty example and then break it down piece by piece.

```javascript
import nme.Manager;
import nme.Timer;
import nme.Point;
import nme.TTF;
class Simple
{
    static var mainObject : Simple;
    var running : Bool;
    static function main()
    {
        mainObject = new Simple();
    }
    public function new()
    {
        var mng = new Manager( 200, 200, "Simple Application", false, "ico.gif" );
        var fps : Float;
        var prevTime : Float = 0.0;
        var curTime : Float;
        running = true;
        while (running)
        {
            mng.events();
            switch mng.getEventType()
            {
                case et_keydown:
                    processKeys( mng.lastKey(), true );
                case et_quit:
                    running = false;
                default:
            }
        }
    }
}
```

(continued)
Part III: Extending the Possibilities

(continued)

curTime = Timer.getCurrent();
fps = 1000.00 / (curTime - prevTime);
prevTime = curTime;
mng.clear( 0x000000 );
TTF.draw( Std.string( fps ), "ARIAL.TTF", 12, new Point( 15, 15 ), 0xFFFF, 0x000000, 100 );
mng.flip();
}
mng.close();
}
public function processKeys( key, pressed : Bool )
{
    switch key
    {
        case 27:
            running = false;
        default:
            neko.Lib.print( key );
    }
}

When running this example, be sure to include the file Arial.ttf in the same directory as the compiled script, as failure to do so may result in the application crashing.

Probably, the most important thing here to notice is the use of the Manager class. This class handles the primary functions that you usually will not be able to do without. This includes creating the window to handle your graphics, clearing or flipping your display buffer, and closing the visible window. If you’re new to games development or multimedia programming outside of Flash, then don’t worry too much if you don’t understand the term flipping your display buffer. You’ll get to that in a moment.

The Manager Class

You’re probably used to working with Neko extensions that provide initialize and cleanup functions. If you compare NME to the various GUI libraries already discussed in this book, then you will be able to note a similar pattern. Like the GUI libraries, NME needs to be setup, then it should enter a loop, and finally, it should be closed down and cleaned up after. The big difference between GUI libraries and NME are that the entered loop is one of your own creations. Most GUI libraries can afford to hide the inner workings of an event loop behind a simple function call. This is because, once the loop is entered, it is quite uncommon for new controls to be created or for existing controls to change properties, with the exception of a change in value. In complete contradiction, a game or multimedia application is expected to change an awful lot between loop iterations, especially with regard to the rendering of graphics. By facilitating your own loop, you can choose exactly what code is run with each iteration enabling complete control over all aspects of your application. Then, each time an iteration draws to an end, you execute a number of functions provided by NME that will handle the physical update onscreen.
Chapter 19: Multimedia with Neko

So, how does this work? Well, first, take a look at the initializing function. In NME, this functionality is assigned to the constructor of the Manager class, so all you have to do is to instantiate it:

```javascript
var width : Int = 300;
var height : Int = 200;
var title : String = "My Window Title";
var fullscreen : Bool = false;
var icon : String = "wndIcon.ico";
var mng = new Manager( width, height, title, fullscreen, icon );
```

Calling the constructor through the instantiation of the Manager class creates a new window object that will contain your application. The parameters accepted by the constructor include the width and height of the new window, the title of the window that will be displayed in its top bar, a Boolean depicting whether the window should be displayed with a frame or expand to fill the entire screen, and the location of an icon file to display in the top bar of the windows frame.

Once you have your Manager instance, you can then use the object to manage the base application, along with the rendering of your graphics. When you finally close the application, you will need to call the close function. This handles the cleanup of the memory used by your application window as well as the sound handler. However, you may also need to individually clean up after your assets, too.

**Flipping the Display Buffer**

When you are working with graphical objects, whether this is your operating system’s native GUI controls, a video feed, or animated content in a multimedia control, the content being displayed is written to the screen display buffer, usually by means of the video card. With GUI controls, this is a smooth process, as the controls aren’t likely to move around much. Likewise, with video, the whole frame is updated at once, so you see little with regard to refresh anomalies. Working with graphical animations, however, can be a little different.

If you write your graphics directly to the display buffer as it is updating, you can often notice tearing, which will appear as an annoying flicker when you run your animation. To get around this, many graphics libraries and platforms provide what is known as double buffering. This means that, instead of writing directly to the display buffer, you write your graphics to a separate off-screen buffer called the back buffer. Then, once you have written all of your graphics, you perform a flip. This performs the task of swapping the back buffer with the current display buffer, creating a nice clean, tear-free refresh. When next you write to the back buffer, you are then writing to what was previously the display buffer. Simple, huh?

NME supports double buffering by default, and there is no way to turn this off. The flipping of the buffers is performed using the flip method of the current Manager object, which you would normally use at the end of a loop block. This will ensure that any data written to the back buffer will be displayed successively.

```javascript
mng.flip{};
```
Part III: Extending the Possibilities

Upon flipping the buffers, the newly appointed back buffer will still contain the artifacts of the previous render. Therefore, before you write the next set of graphics to the buffer, it’s often best to perform a wipe of the back buffer, first. You do this using the clear method of the Manager class, passing the color you’d like it cleared with, as represented by a hex value.

```javascript
var bgColor : Int = 0x000000;
mng.clear( bgColor );
```

**Slowing the Rate of Refresh**

If you ran the previous example, you should have been presented with a small window showing a numeric value in the top left, as shown in Figure 19-1.

![Figure 19-1](image-url)

This value represents the current number of frames per second being drawn to the display buffer. When we ran this example on our laptop, we managed to achieve around 250 frames per second maximum, though in all likelihood, we probably exceeded this somewhat. Now, when you come to write an application with NME, you’re not likely to want to have your graphics updated at this speed. For most, this is far too fast and will result in a comical high-speed animation effect that will render most games unplayable. To avoid this, the Manager class provides a method called delay, which is a means to pause the application for a set number of milliseconds, allowing for a reasonable slowdown in frame rate.

```javascript
var milliseconds : Int = 1 / preferredFPS;
mng.delay( milliseconds );
```

When you use this method, the operating system leaves the current thread of execution, allowing for other applications running at that time to perform their processing. The exact number of milliseconds the delay will last will likely vary by around 10 milliseconds, because of the nature of thread execution. Now, a word of warning when using delay; it is usually better to use this method when running your
application full screen. The reason for this is that it can cause a few glitches with the rendering of your other applications, sometimes causing the NME program to crash. If you need to run your NME application in a window, try using a Timer to manage the speed of your loop, instead. Timers are discussed later in this chapter.

Writing Text to the Display

Text is written to the display device using literal font files, which need to be stored somewhere in the applications directory structure. In most of the examples in this chapter where text is used, an Arial.ttf file is used in the same location as the executed script file. Failure to find this file when running your application may cause it to crash. We hope to have this resolved in a future release of NME.

Text in applications plays a big part, though using text in NME is extremely simple. So simple in fact, that the class that deals with writing text to the display, the TTF class, only has one single method.

```javascript
var textStr : String = "Hello, World!";
var font : String = "Arial.ttf";
var ptSize : Int = '12';
var location : Point = new Point( 10, 10 );
var fgColor : Int = 0xFFFFFF;
var bgColor : Int = 0x000000;
var alpha : Int = 100;
TTF.draw( textStr, font, ptSize, location, fgColor, bgColor, alpha );
```

The parameters are fairly self-explanatory. The first parameter is the string you want written to the Surface. This can be as long as you like, though the window only has so much real estate. From there, the parameters deal with the font file you wish to use, the size of the font to display on screen in pixels, the location to print the text as a Point instance, the foreground and background colors, and the alpha value of the displayed text, respectively. When choosing colors for the text, the background has to be rendered to screen, unfortunately, though you can avoid this issue using color keys described later in this chapter.

Working with Surfaces

All data written to the display buffer, whether graphical or textual, is done so using a Surface object. Surfaces provide a great way to interlink all assets in your NME applications, and allow for rendering to be performed on any visible object, seeing as the Surface class represents the only visible object type. Even the display buffer is represented in NME as a Surface. This means you can just as simply render all of your text and animations to an object, and then render that object to the display buffer, if you so wish. Or, you could choose to render the content of the back buffer to an area on a different surface. Though, quite frankly, we couldn’t imagine a time when this would be useful. Figure 19-2 represents the relationship between each of the classes in NME and the Surface class.
Part III: Extending the Possibilities

The `Surface` class provides a number of functions relative to interacting with, or manipulating itself. Functionality such as transforms, collision detection, and color keying, can be found here, though most interaction with a `Surface` object will normally be handled from the `Sprite`, `Manager`, or one of the font classes.

To use a `Surface`, you first need to provide it with an image, which can be a BMP, PNM (PPM/PGM/PBM), XPM, LBM, PCX, GIF, JPEG, PNG, TGA, or TIFF format. This needs to be passed when you instantiate the class as a path to the image file.

```javascript
var mySprite : String = "assets/charSprite.bmp";
var srf = new Surface( mySprite );
```

The image is then stored in memory and can be accessed through the `Surface` object instance. When you are finished with the image, you should call the method `free`, which relinquishes the memory consumed by the image.

```javascript
srf.free();
```

This is not necessary, however, when ending your application providing you remember to call the `close` method of the `Manager` class, as all remaining `Surface` objects are cleared at this time.

**Drawing a Surface to the Display**

The `Surface` class provides a simple method called `draw`. This method is responsible for copying a series of pixels from itself to a location on another `Surface` object, which can be either another image, or the back buffer of the display.

```javascript
var destination : Surface = mng.getScreen();
var pixels : Rect = new Rect( 0, 0, 100, 100 );
var location : Point = new Point( 0, 0 );
srf.draw( destination, pixels, location );
```
Chapter 19: Multimedia with Neko

The pixels value represents the coordinates of the pixels to draw to the destination Surface, which include both the x and y location, as well as the width and height of the pixel area. The location of the destination Surface is merely the x and y values, as the pixels are transferred without transformation. You do not have to specify the source Surface, as it is assumed you wish to draw from the methods object.

To draw to the display buffer, you first need to acquire its Surface object. This is performed using the getScreen method of the Manager class instance.

```javascript
mng.getScreen();
```

Once acquired, you can then access the Surface as you would any other, whether you are drawing to, or from its pixels.

Here is an example utilizing the draw method:

```javascript
import nme.Manager;
import nme.Surface;
import nme.Rect;
import nme.Point;
import nme.TTF;
class DrawSurface
{
    static var mainObject : DrawSurface;
    var running : Bool;
    var mng : Manager;
    var batSrf : Surface;
    static function main()
    {
        mainObject = new DrawSurface();
    }
    public function new()
    {
        mng = new Manager( 200, 200, "Surface Draw", false, "ico.gif" );
        batSrf = new Surface( "bat.png" );
        batSrf.setKey( 0xFF, 0x00, 0xFF );
        var x = 30;
        var y = 30;
        var dir = true;
        running = true;
        while (running)
        {
            mng.clear( 0x00000000 );
            batSrf.draw(Manager.getScreen(), new Rect(24, 63, 65, 44), new Point(x, y));
        }
    }
}
```
Part III: Extending the Possibilities

```java
if ( dir == true )
    x = y = x + 10;
else
    x = y = x - 10;
if ( x > 150 ) dir = false;
if ( x < 40 ) dir = true;
mng.flip();
mng.delay( 40 );
}
batSrf.free();
mng.close();
}
```

*Make sure to include the bat.png file, or the application will fail.*

### Setting the Color Key

When drawing the rect of an image to a `Surface`, you often won’t want to draw every single pixel. For certain, most game characters aren’t square, and as such, you’ll need a way to remove unwanted pixels from your image. This is done using a technique called *keying*.

Keying is a means to provide a mask whereby any pixel that appears within the mask criteria are not blitted (Bit Block Transfer) to the destination surface. This sort of technique can be seen in film using chroma-keying, where footage is superimposed onto another while first having all areas of a certain color removed.

The previous example performs the feat of keying using the `setKey` method of the `Surface` instance. This method accepts the red, green, and blue values of the color you wanted masked as 8-bit values (0 to 255).

```java
var red : Int = 0xFF;
var green : Int = 0x0;
var blue : Int = 0xFF;
srf.setKey( red, green, blue );
```

When working with image formats that support an alpha channel, remember that you can use this alpha channel to define your character shapes. Try experimenting with alpha channels to get the effect that works best.

### Surface Transforms

The NME library provides three functions for transforming the image on a surface. Unfortunately, all three of these functions work with the entire surface, though there are ways around using individual images for each frame if you know what you’re doing.

Two of the three functions used for transformation of `Surfaces` provide a distortion of the pixels. Both of these functions perform a rotate and a scale on the source `Surface`. The first of these transform functions, `transform`, has this signature.
Chapter 19: Multimedia with Neko

```javascript
var screen : Surface = mng.getScreen();
var angle : Float = 90.0;
var scale : Point = new Point( 1, 1 );
var pivot : Point = new Point( 50, 50 );
var destination : Point = new Point( 0, 0 );
var flags : Int = Surface.DEFAULT;
var renderedArea : Rect = srf.transform( screen, angle, scale, pivot, destination, flags );
```

The `screen` parameter is the private `srf` variable in the `Surface` class or the returned value of the `getScreen` method of the `Manager` class. If using a `Surface`, you will need to prepend the transfer using the `untyped` keyword, so as to please the compiler. This value is the raw C struct representing the image in memory.

The `angle` parameter is a float value from 1 to 360 and represents the angle of rotation in degrees. If you do not want to apply a rotation, you can simply pass 0. The `pivot` parameter is paired with this value, and represents where on the source image the rotation should occur, as contained in a `Point` class instance.

The third parameter, `scale`, represents the amount at which the image should be, well, scaled. Passing a positive integer for either `Point` value will respond with an enlargement for that dimension, while a negative integer for either `Point` value will respond with a reduction.

Finally, the `destination` parameter is the location on the destination `Surface` as a `Point` object, while the `flags` parameter accepts one of the static variables of the `Surface` class, as detailed in Table 19-1.

**Table 19-1**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT</td>
<td>The default value. Performs no further alteration.</td>
</tr>
<tr>
<td>TAA</td>
<td>Uses the interpolating renderer, which is much slower but can look better.</td>
</tr>
<tr>
<td>TSAFE</td>
<td>Doesn’t assume that the source and destination surfaces have the same pixel format. This is the default when the two surfaces don’t have the same bits per pixel and is slower but will render weird pixel formats properly.</td>
</tr>
<tr>
<td>TTMAP</td>
<td>Use texture mapping. This is a bit faster but the result isn’t as nice as the default mode. This mode will also ignore the pivot coordinates and any other flags used.</td>
</tr>
</tbody>
</table>
Part III: Extending the Possibilities

The `transform` method returns a `Rect` object detailing the size and location of the image drawn on the destination `Surface` object. You can use this to for further function processing, such as collision detection.

The following example shows the `transform` method in use, along with one of the ways to avoid using a single image `Surface` object:

```java
import nme.Manager;
import nme.Surface;
import nme.Rect;
import nme.Point;
import nme.TTF;

class TransformSurface
{
    static var mainObject : TransformSurface;
    var running : Bool;
    var mng : Manager;
    var batSrf : Surface;
    var dispSrf : Surface;
    static function main()
    {
        mainObject = new TransformSurface();
    }
    public function new()
    {
        mng = new Manager( 200, 200, "Surface Draw", false, "ico.gif" );
        batSrf = new Surface( "bat.png" );
        dispSrf = new Surface( "bat.png" );
        dispSrf.setKey( 0xFF, 0x00, 0xFF );
        var dir = true;
        var angle = 0;
        running = true;
        while (running)
        {
            mng.clear( 0x00000000 );
            dispSrf.clear( 0xFF00FF );
            batSrf.transform( untyped dispSrf.__srf, angle, new Point(1, 1),
                             new Point(56, 87), new Point(70, 70), Surface.DEFAULT );
            dispSrf.draw( Manager.getScreen(), new Rect(35, 35, 90, 70),
                          new Point( 70, 70 ) );
            if ( dir == true )
                angle += 10;
            else
                angle -= 10;
            if ( angle > 350 ) dir = false;
            if ( angle < 10 ) dir = true;
            mng.flip();
            mng.delay( 40 );
        }
    }
}
```
The alternative to the transform function is the `transformSurface` method. This method differs only slightly, and provides the following signature:

```javascript
var bgColor : Int = 0x000000;
var angle : Float = 90.0;
var scale : Point = new Point( 0.5, 0.5 );
var flags : Int = Surface.DEFAULT;
srf.transformSurface( bgColor, angle, scale, flags );
```

Here, instead of drawing the resulting rotation to a `Surface`, it is instead returned on a new `Surface` object, leaving the original `Surface` unchanged. The new parameter, `bgColor`, is the color of the new `Surface` in general where the transformed `Surface` is not drawn. One would usually opt to use a color that can be keyed for this parameter.

The remaining function for performing transforms is the `setAlpha` method. As expected, `setAlpha` alters the alpha value for the entire image to between 0 and 100 percent. The result then replaces the image in memory.

```javascript
var percentage : Int = 100;
srf.setAlpha( percentage );
```

### Detecting Collisions

The `Surface` class has two methods that are useful for detecting collisions with other `Surfaces`. The difference between the methods is the level of accuracy, where one of the functions, `collisionBox`, performs a simple bounding box detection, while the other, `collisionPixel`, performs a pixel perfect detection. When choosing which type of collision detection to utilize, remember first that the `collisionPixel` method is by far the slower, as each pixel has to be individually checked within both `Surfaces` before a match is found. `collisionBox`, on the other hand, is a simple mathematical query, and so takes minimal calculation. It is also worth noting that the `collisionPixel` method automatically performs a `collisionBox` calculation in order to reduce possible redundant calls.

To use `collisionBox`, you merely have to pass the `Rect` values of the two colliding `Surfaces`, along with a `Point` value depicting where on the primary `Surface` (the `Surface` that makes the collision check) the secondary `Surface` begins.

```javascript
var srcRect : Rect = new Rect( 0, 0, 20, 20 );
var destRect : Rect = new Rect( 40, 40, 20, 20 );
var offsetPoint : Point = new Point( 10, 10 );
var hasCollided : Bool = srf.collisionBox( srcRect, destRect, offsetPoint );
```

If a collision is detected, then the method call returns `true`; otherwise `false` is returned.
Part III: Extending the Possibilities

The `collisionPixel` method uses almost the same routine. However, as the individual pixels of both `Surfaces` need to be analyzed, the `collisionPixel` method also requires the actual `Surface` image for the calculation.

```javascript
var srf2 : Surface = new Surface( "assets/charSprite2.bmp" );
var srcRect : Rect = new Rect( 0, 0, 20, 20 );
var destRect : Rect = new Rect( 40, 40, 20, 20 );
var offset : Point = new Point( 10, 10 );
var hasCollided : Bool = srf.collisionPixel( srf2, srcRect, destRect, offset );
```

The following is an example of collisions in use. Drag the bat in the top left corner over the bat in the center. If a bounding box collision is made, the screen will flash orange, while a pixel-based collision will cause the screen to flash red.

At this point in time, do not worry a great deal about the Mouse or Event features used in this example. They will be discussed in detail a little later in the chapter.

```javascript
import nme.Manager;
import nme.Surface;
import nme.Rect;
import nme.Point;
import nme.TTF;
class SimpleCollision
{
    static var mainObject : SimpleCollision;
    var running : Bool;
    var click : Int;
    var batR : Rect;
    var bat : Point;
    var bat2 : Point;
    var batSrf : Surface;
    var curTime : Float;
    var prevTime : Float;
    static function main()
    {
        mainObject = new SimpleCollision();
    }
    public function new()
    {
        prevTime = 0;
        curTime = 0;
        click = 0;
        var mng = new Manager( 200, 200, "Collision Test", false, "ico.gif" );
        batSrf = new Surface( "bat.png" );
        batR = new Rect(24, 63, 65, 44);
        bat = new Point( 0, 0 );
        bat2 = new Point( 50, 50 );
        batSrf.setKey( 255, 0, 255 );
        var fps : Float;
        running = true;
    }
}
```
while (running)
{
    mng.events();
    switch mng.getEventType()
    {
    case et_mousebutton:
        if ( mng.mouseButton() == 1 )
            if ( mng.mouseButtonState() == 1 )
            {
                if ( mng.clickRect( mng.mouseX(), mng.mouseY(), new Rect(bat.x, bat.y, 
                                batR.w + bat.x, batR.h + bat.y) ) )
                    click = 1;
                else
                    click = 0;
            }
            else
            {
                click = 0;
            }
    case et_mousemove:
        if ( click == 1 )
        {
            bat.x += mng.mouseMoveX();
            bat.y += mng.mouseMoveY();
        }
    case et_quit:
        running = false;
    default:
    }
    if( batSrf.collisionPixel( batSrf, batR, batR, new Point( bat.x - bat2.x, 
                                bat.y - bat2.y ) ) )
        mng.clear( 0xFF0000 );
    else if( batSrf.collisionBox( batR, batR, new Point( bat.x - bat2.x, 
                                bat.y - bat2.y ) ) )
        mng.clear( 0xFF9900 );
    else
        mng.clear( 0x00000000 );
    batSrf.draw(Manager.getScreen(), batR, bat);
    batSrf.draw(Manager.getScreen(), batR, bat2);
    mng.flip();
}
batSrf.free();
mng.close();
}
Part III: Extending the Possibilities

Keeping Time with Timers

Programming games is a resource consuming business. So far, all of the examples depicted in this chapter have made use of the delay method to help slow down the rate of animation, but this is certainly not the most effective option, and when running in a window, can seem a little buggy. Ask any game developer, and they’ll tell you that it’s okay to hoard as much of a machine’s resources as you can, within reason of course. While deferring control of a thread to the operating system may be good practice when GUI programming, which is what you are effectively doing with the delay method, this isn’t the right code of conduct when developing games. The reason for this is that it’s an unreliable method. You never really know if the operating system will return with due haste to your application, which can make things seem sluggish and unplayable.

The way to avoid this issue is to use a Timer object. Timers are very simple things, but they allow you to calculate the amount of time that has passed, and with such knowledge, you can personally manage whether to execute a sequence of events, or not.

One would normally create a Timer object for a repetitive procedure that needs to recur every so many milliseconds. This may be different for many processes within a game, so multiple Timers can come into play at once. Luckily, Timers use minimal resources; certainly a lot less than it costs to defer a thread with delay.

```plaintext
var milliseconds : Int = 1 / preferredFPS;
var timer : Timer = new Timer( milliseconds );
```

Once a Timer is in use, you can check whether the required amount of time has passed by querying its isTime method. Now, in an ideal world, the Timer would fire an event when such a time has passed, but seeing as games aren’t usually event based, you aren’t offered that luxury. The isTime method will return true if the set amount of time comes to pass.

For ad hoc occasions, one could query the getCurrent method to acquire the number of milliseconds passed since the NME library was initialized, while the getPrevious method returns the number of milliseconds since isTime last returned true.

The Wonderful World of Sprites

Now you’re getting to the fun stuff. Sprites, managed by the Sprite class, represent the animation control of the graphical assets in your applications. With these babies, you can do just about anything, from a simple game character to a business presentation. It’s all possible.

A sprite is normally a representation of one or more images that form an animated character. This series of animation frames usually derives from a single image, or sprite sheet. The sprite sheet displays a number of frames in a grid with each frame being of the same size for each animation sequence. Figure 19-3 depicts a typical sprite sheet, courtesy of Ari Feldman (www.flyingyogi.com).
Acquiring a Sprite from a Sprite Sheet

By far the easiest, and probably the least memory hungry way of using a sprite sheet is to load the entire sheet into memory, then display only the segment you wish to show at any one time. This will give the impression of swapping frames from many images, while only having to keep track of one image.

The NME library performs this task by storing the coordinates of each frame as an array of four integer values stored in an object that represents the rectangle of pixels to display. This array is further broken down into an array of animation sequences, as a sprite sheet will often represent more than a single sequence.

The frames of a sequence can be set manually, one at a time, using the `setFrame` method.

```javascript
var frame : Rect = new Rect( 0, 0, 16, 16 );
var group : Int = 0;
var location : Int = 0;
sprite.setFrame( frame, group, location );
```
Part III: Extending the Possibilities

Here, the frame parameter is a Rect object representing the location and size of the image within the sprite sheet.

The second parameter, group, represents the sequence of animation. For example, within a single sprite sheet, we might have fifteen different sequences, whereby my character will perform different moves. A group is just that; a single sequence of frames representing a single move, topic, or other such related behavior.

Finally, the location parameter. This is merely the position in the group array that the frame should appear. Normally this would be the position after the last frame that already exists in the group, but can vary, particularly if you are replacing a frame.

An example of frames being added for the bat character of the previous code might look like this.

```
bat.setFrame( new Rect(24, 63, 65, 44), 0, 0 );
bat.setFrame( new Rect(156, 63, 65, 44), 0, 1);
bat.setFrame( new Rect(288, 63, 65, 44), 0, 2 );
bat.setFrame( new Rect(420, 63, 65, 44), 0, 3 );
```

As you can see, here the images are defined for the locations 0 through 3 of group 0.

An alternative, and indeed, a much quicker option, is to use the setFrameRange method.

```
var xOffset : Int = 0;
var yOffset : Int = 0;
var sprWidth : Int = 16;
var sprHeight : Int = 16;
var columns : Int = 6;
var count : Int = 20;
var group : Int = 0;
srf.setFrameRange( xOffset, yOffset, sprWidth, sprHeight, columns, count, group );
```

This method performs the task of adding multiple frames at once to a group, by defining the start location of the frame range on the sprite sheet with the xOffset and yOffset parameters, then bit by bit, iterating through each frame until count frames have been added. The setFrameRange method continues horizontally across the sprite sheet until it has added frames to a multiple of the cols parameter, before returning and continuing from a new line. The size of each frame, and indeed, the height of each row of frames, is depicted with the spriteWidth and spriteHeight parameters. Once all frames have been calculated, they are stored in the given group location.

One point to notice with regard to the setFrameRange method is that it doesn’t expect any empty space to exist between frames, either horizontally or vertically. This, however, may change with future version of NME.
Chapter 19: Multimedia with Neko

**Animating Sprites**

Animating the Sprite object actually makes drawing to a Surface a lot easier. Providing you have all of your frame locations calculated, you can begin rendering your animation with a simple call to the animate method. This method, unlike the draw method of the Surface class, accepts a single parameter: a Timer instance. The Timer is used to calculate when the next frame in a given sequence should be rendered.

Alongside the animate method, you can also choose how your animation should flow. Like animate, choosing the sequence that your frames are rendered is incredibly simple, and is carried out by setting a type parameter of the Sprite class using an AnimType enumerator. The type parameter depicts whether the animation will play once, continuously loop, or continuously ping pong from the first frame to the last and back again. Each one is represented by the at_once, at_loop, and at_pingpong enumerators, respectively.

When your animation is running, you may need to change the animated sequence entirely. You can do this by setting the group parameter to the group id of the sequence you wish to render. You may also want to change the type parameter at this time.

At any time during an animation, or indeed when the sprite is static, you can opt to query or set the current frame of the animation by modifying the currentframe property. Likewise, you can also alter the location the Sprite is drawn to the destination Surface, by modifying the x and y properties.

**Interaction Through Events**

Okay, so you've seen how you can render to the screen using Surfaces and Sprites, but what about interacting with those objects? NME provides quite an array of features for interaction, whether with a mouse or keyboard. At present, joystick support is not available in NME, though this is planned for the near future.

**Capturing Events**

Unlike GUI programming, events are not fired, and they don't use callbacks. Instead, the events object of the underlying SDL library needs to be requested in order to bring it up to date with the running application, and then you can query it bit by bit. Requesting events is merely a way to clear and fill an event stack with any event objects since the event object was last requested. Once the request is complete, you can then query the stack for the events you are most interested in, and then act upon them in your application. This is probably akin to how a GUI events system works, only in an as requested fashion, rather than an enforced one.

To request the events to be gathered, you make a call to the events method of an instantiated Manager object. Once performed, you can then query the events stack by type using the plethora of event-related methods, also within a Manager instance. Table 19-2 details these methods.
Part III: Extending the Possibilities

Table 19-2

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetEventType</td>
<td>Pops an event off of the event stack and returns an EventType enumerator</td>
</tr>
<tr>
<td></td>
<td>depicting the type of the event. Once popped, you can then query the even</td>
</tr>
<tr>
<td></td>
<td>further using one of the methods below.</td>
</tr>
<tr>
<td>LastKey</td>
<td>Returns the ASCII key code of the last pressed key. You can use the haXe</td>
</tr>
<tr>
<td></td>
<td>String class method fromCharCode to convert the value back into a</td>
</tr>
<tr>
<td></td>
<td>recognizable character.</td>
</tr>
<tr>
<td>MouseButton</td>
<td>Returns the last clicked mouse button, where 1 is the left button, 2 is the</td>
</tr>
<tr>
<td></td>
<td>right button, and 3 is the middle button.</td>
</tr>
<tr>
<td>mouseButtonState</td>
<td>Returns the pressed state of the last pressed button, where 1 is pressed</td>
</tr>
<tr>
<td></td>
<td>and 0 is released.</td>
</tr>
<tr>
<td>MouseX</td>
<td>Returns the current x coordinate of the mouse cursor.</td>
</tr>
<tr>
<td>MouseY</td>
<td>Returns the current y coordinate of the mouse cursor.</td>
</tr>
<tr>
<td>mouseMoveX</td>
<td>Returns the x coordinate distance travelled of the mouse cursor since the</td>
</tr>
<tr>
<td></td>
<td>last query.</td>
</tr>
<tr>
<td>mouseMoveY</td>
<td>Returns the y coordinate distance travelled of the mouse cursor since the</td>
</tr>
<tr>
<td></td>
<td>last query.</td>
</tr>
</tbody>
</table>

The following example is an update of the SimpleCollision class, but with updates detailing animation, timers, and events:

```plaintext
import nme.Manager;
import nme.Surface;
import nme.Sprite;
import nme.Rect;
import nme.Point;
import nme.Timer;
import nme.TTF;
class Collision
{
    static var mainObject : Collision;
    var running : Bool;
    var bat : Sprite;
    var bat2 : Sprite;
    var keys : Array<Bool>;
    var curTime : Float;
    var prevTime : Float;
    static function main()
```
Chapter 19: Multimedia with Neko

```javascript
{
    mainObject = new Collision();
}
public function new()
{
    keys = new Array();
    prevTime = 0;
    curTime = 0;
    var mng = new Manager( 200, 200, "Collision Test", false, "ico.gif" );
    var batSrf : Surface = new Surface( "bat.png" );
    bat = new Sprite( batSrf );
    bat2 = new Sprite( batSrf );
    batSrf.setKey( 255, 0, 255 );
    bat.setFrame( new Rect(24, 63, 65, 44), 0, 0 );
    bat.setFrame( new Rect(156, 63, 65, 44), 0, 1 );
    bat.setFrame( new Rect(288, 63, 65, 44), 0, 2 );
    bat.setFrame( new Rect(420, 63, 65, 44), 0, 3 );
    bat2.setFrame( new Rect(24, 63, 65, 44), 0, 0 );
    bat2.setFrame( new Rect(156, 63, 65, 44), 0, 1 );
    bat2.setFrame( new Rect(288, 63, 65, 44), 0, 2 );
    bat2.setFrame( new Rect(420, 63, 65, 44), 0, 3 );
    bat.type = at_loop;
    bat.group = 0;
    bat2.type = at_pingpong;
    bat2.group = 0;
    bat2.x = 60;
    bat2.y = 60;
    var iTimer : Timer = new Timer( 5 );
    var jTimer : Timer = new Timer( 7 );
    var kTimer : Timer = new Timer( 5 );
    var gTimer : Timer = new Timer( 25 );
    var fps : Float;
    running = true;
    while (running)
    {
        mng.events();
        switch mng.getEventType()
        {
            case et_keydown:
                processKeys( mng.lastKey(), true );
            case et_keyup:
                processKeys( mng.lastKey(), false );
            case et_mousebutton:
                if ( mng.mouseButton() == 1 )
                    if ( mng.mouseButtonState() == 1 )
                    {
                        var tmp : Rect = bat.getCurrentRect();
                        var batRect : Rect = new Rect( bat.x, bat.y, tmp.w, tmp.h );
                        if ( mng.clickRect( mng.mouseX(), mng.mouseY(), batRect ) )
                            bat.click = 1;
                }
```

(continued)
Part III: Extending the Possibilities

(continued)

```java
else
    bat.click = 0;
} else {
    bat.click = 0;
}

case et_mousemove:
    if (bat.click == 1) {
        bat.x += mng.mouseMoveX();
        bat.y += mng.mouseMoveY();
    }

case et_quit:
    running = false;

default:
}

curTime = Timer.getCurrent();
if (kTimer.isTime())
    fps = 1000.00 / (curTime - prevTime);
prevTime = curTime;

if (gTimer.isTime()) {
    if (keys[0]) bat.y -= 1;
    if (keys[1]) bat.y += 1;
    if (keys[2]) bat.x -= 1;
    if (keys[3]) bat.x += 1;
    if (batSrf.collisionPixel(batSrf, bat.getCurrentRect(), bat2.getCurrentRect(), bat.getSpriteOffset(bat2)))
        mng.clear(0xFF0000);
    else if (batSrf.collisionBox(bat.getCurrentRect(), bat2.getCurrentRect(), bat.getSpriteOffset(bat2)))
        mng.clear(0xFF9900);
    else
        mng.clear(0x00000000);
    bat.animate(iTimer);
    bat2.animate(jTimer);
    mng.flip();
}
}
batSrf.free();
mng.close();
}

public function processKeys(key, pressed : Bool) {
    switch key {
    case 27:
        running = false;
    case 273:
        keys[0] = pressed;
    ```
case 274:  
    keys[1] = pressed;  
  case 275:  
    keys[3] = pressed;  
  case 276:  
    keys[2] = pressed;  
  default:  
    neko.Lib.print( key );

As you can see, in this particular example, everything is used with the exception of the setFrameRange method of the Sprite class. This is because of the spacing anomalies within the given sprite sheet that these methods aren’t designed to handle.

Summary

This chapter portrayed one of the fun areas of programming with haXe and Neko. If you already dabble in games and multimedia development, then much of this chapter should not have been too difficult a venture, while those new to the concept of games development may want to read a little further into the general concepts of programming game loops and logic.

In this chapter, you learned:

- Why Neko is great for games development
- The differences between the NME and nGame libraries
- How to create a simple games loop
- How to work with display buffers and double buffering
- How to use Surfaces
- How to animate with Sprites
- How to keep time with Timers
- How to deal with user interaction and events

In the next chapter, you learn how to create your own Neko extensions, such as NME, using C or C++.
Extending haXe with C/C++

The Neko Virtual Machine is very powerful, and the complete haXe framework does provide an incredibly large number of features suitable for most application requirements. That doesn't mean, however, that it caters to all of your programming needs. If you require features in haXe for your desktop or server-side logic, you prefer specific optimized code, or you simply like extending languages, then you'll need to get your hands dirty with a little C or C++ development. In this chapter, you'll do just that by taking a look at the following:

- What required files are needed by the compiler for building Neko libraries
- How to expose C/C++ functions in a dynamic library for inclusion in Neko
- How to convert data types between Neko and C/C++
- How to deal with pointers to unsupported Neko types
- How to handle errors in your code
- How to make explicit use of the garbage collector

Neko Libraries

To facilitate custom libraries in C/C++, Neko provides a C foreign function interface, or C FFI. It is the purpose of this C FFI to expose all the necessary functionality within a C/C++ dynamic link library to provide a usable relationship between the two (or indeed three) languages.

All Neko libraries are really dynamic link libraries written for the Neko Virtual Machine using the extension ndll, instead of the more familiar dll. This is the same for all supported operating systems, so as to maintain uniformity. If you take a look in the Neko directory, you will see that several libraries with this extension are available with the initial Neko installation. When creating your first Neko library, you can use these preexisting libraries in order to gain a better understanding for creating the various haXe classes that will wrap your ndll functionality. For example, the regexp ndll library is wrapped with the EReg.hx class in the root of the haXe
Part III: Extending the Possibilities

standard framework. Also, if you’re feeling really adventurous, it might pay to download the source for
the Neko Virtual Machine and its tools and check out firsthand all of the inner workings of a fully
fledged Neko library.

Wrapping Neko ndll files will be covered later in this chapter.

**Setting Up the Compiler**

If you’ve never coded in C or C++ before, then it is suggested to read a good book on the subject, first.
C and C++ are both complex languages, and their use is beyond the scope of this book.

It is not assumed there is any particular orientation to a specific compiler, simply because, if you’re
reading this chapter, then you will probably already have a preference for a particular compiler, and at
least have some grounding in how to use it. Luckily, though, setting up a compiler for Neko libraries is
incredibly easy, so even a relative novice should have little trouble.

Like most C/C++–based libraries, Neko provides a library file for inclusion into your projects that will
provide all the functionality necessary for building your extension. On Windows machines, this is called
neko.lib, Linux has libneko.so, and Mac OSX provides libneko.dylib. Depending on the
particular operating system, you can find this file in either the Neko directory or in the global library
repository. Upon creating your new project, you will need to provide a reference to this file in the
projects makefile or compiler settings. You will also need to specify a reference to the neko.h header
file found in the Neko include directory, so that the functions and type definitions contained in the
library file can be validated at compile time.

**A Simple Hello World Example**

Okay, so a Hello World example might be a little too simple to be anything useful, especially to all the
hard-core C/C++ developers, but it will make a great starting point. Let’s jump in to one such example,
now. Work through the following steps, producing a complete, albeit useless, Neko module. You’ll get to
the part about how each fragment works afterward.

**Building the ndll Library in C**

The following code describes a standard Hello World example in C with the necessary commands
needed to make it Neko compatible:

```c
#include "neko.h"
#include <stdio.h>
value PrintHelloWorld()
{
    printf( "Hello, World!" );
    return val_null;
}
DEFINE_PRIM( PrintHelloWorld, 0 );
```
This is about as simple as it gets, and indeed, should closely resemble a Hello World example in any C/C++ book. Compile this as hello.ndll, and place the output file into the Neko directory. You can now use this library from either a haXe script or a Neko script. If you want, you could instead place the library in a working haXe directory for one of your projects. Either way, the Neko runtime will be able to locate the library and execute the specified function.

If you receive errors while compiling this code, be sure to check the references for the included directory and the path to the Neko library file. This will usually form a majority of the issues found by developers new to C/C++.

Building the haXe Script

To access the PrintHelloWorld function in your library from within a haXe script, create a new haXe file and enter the following:

```haXe
class HelloWorld
{
    public static function main()
    {
        hello();
    }
    static var hello = neko.Lib.load( "hello", "PrintHelloWorld", 0 );
}
```

Now, compile the haXe file and execute it from a command prompt. The script will run and immediately exit, presenting the text:

```
> Hello, World!
```

The Post-Mortem

While this example doesn’t demonstrate much of the Neko C foreign function interface, it does outline its simplicity. If you’ve ever tried extending a scripting language before, you might agree that it’s not always a very intuitive process. Of all the languages we’ve ever had the pleasure of dealing with, whether it was Python, Ruby, TCL, or Lua, none of them were considerably easy or clear. Neko, however, is both pretty and simple, while providing powerful routines for managing data conversion and execution between both languages.

If you examine both the C program code and the haXe class used in the previous example, you should immediately be able to notice what is going on. In the C code, the `DEFINE_PRIM` macro serves the purpose of exposing the passed function pointer to the Neko Virtual Machine outside of the compiled library. The second parameter of this macro states the number of parameters that this function requires.

```c
DEFINE_PRIM( FunctionPointer, intNumParams );
```

In the haXe class, you then have a static variable that accepts a pointer to this function, so that it might be executed directly within the Neko compiled code.

```haXe
static var funcPtr = neko.Lib.load( "LibraryName", "FunctionName", intNumParams );
```
Part III: Extending the Possibilities

The variable is set as static so that it receives the function pointer without its parent class requiring instantiation. While this is not the only way to do this, it is the most common method and is used by all of the Neko libraries included with the basic Neko installation. However, as all the static C function pointer variables are set at the time when the Neko script initializes, setting more than a few hundred of these variables in one go may cause the virtual machine to bend under the strain, if not crash completely. At such times, it is often better to assign such references when they are needed, rather than all at once.

Neko Value Types in C/C++

So, you’ve seen how simple it is to get a quick library up and running, and how to call a function within this library, but what about passing and receiving variables? Before you go headfirst into this topic, it would first help to understand the structure of Neko values.

The Neko value Struct

If you recall from Chapter 3 when haXe types were discussed, it was noted that all Neko base value types were defined internally as a C struct containing two values, the data of the value and the type of the value. Dealing with values in this way enables the Neko run time to work at an optimized rate, while providing extremely flexible extensibility to the developer wishing to extend the language.

```
struct value {
  int value_type;
  void *value_data;
};
```

The value struct is a great way to seamlessly pass data from Neko to C/C++ and back again. This is because, although absolutely any data type can be contained within the struct, both C/C++ and Neko have a way to assume the data type where possible. Also, passing data to functions where all data types are known as value means that absolutely any data type can be passed as a function parameter if so wished. Though of course, without proper error checking in each of your C/C++ and haXe functions, you may be setting yourself up for trouble.

Take a look again at the HelloWorld example. This time, the C/C++ function will accept a string to be displayed:

```
#include "neko.h"
#include <stdio.h>
value PrintHelloWorld( value str )
{
  val_check( str, string );
  printf( val_string( str ) );
  return val_null;
}
DEFINE_PRIM( PrintHelloWorld, 1 );
```
And again, with the modified haXe class:

```haXe
class PrintHelloWorld
{
    public static function main()
    {
        hello( neko.Lib.haxeToNeko( "Hello, World!" ) );
    }
    static var hello = neko.Lib.load( "hello", "PrintHelloWorld", 1 );
}
```

As you can see within the C/C++ code, the parameter `str` is passed to the function `val_string()`. This function is necessary to extract the data contained inside the `value` struct. There is one of these functions for each value type supported by the Neko Virtual Machine, and each are aptly named. Table 20-1 details each function in turn.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val_int( value v )</td>
<td>Returns the integer value stored in the value struct</td>
</tr>
<tr>
<td>val_float( value v)</td>
<td>Returns the float value stored in the value struct</td>
</tr>
<tr>
<td>val_bool( value v)</td>
<td>Returns the Boolean value stored in the value struct</td>
</tr>
<tr>
<td>val_string( value v)</td>
<td>Returns the string value stored in the value struct</td>
</tr>
<tr>
<td>val_strlen( value v)</td>
<td>Returns the length of the string value stored in the value struct</td>
</tr>
<tr>
<td>val_array_ptr( value v)</td>
<td>Returns the array stored in the value struct as a pointer of type value</td>
</tr>
<tr>
<td>val_array_size( value v)</td>
<td>Returns the size of the array stored in the value struct</td>
</tr>
<tr>
<td>val_fun_nargs( value v)</td>
<td>Returns the number of arguments required by the function stored in the value struct</td>
</tr>
<tr>
<td>val_data( value v)</td>
<td>Returns the abstract value stored in the value struct</td>
</tr>
<tr>
<td>val_kind( value v)</td>
<td>Returns the abstract constant identifier stored in the value struct</td>
</tr>
</tbody>
</table>

### Preparing Data for Transit from haXe to C/C++

If you look closely at the haXe class in the preceding example, the string value being passed to the C/C++ function also required wrapping inside a helper function, `neko.Lib.haxeToNeko`. The reason for this does not have to do with the differences between Neko and C/C++, but between Neko and haXe. In Neko, each value type used in a Neko script can be safely and directly used by C/C++, albeit sometimes with the help of a C function. However, some of the types native to haXe do not map directly to Neko types as they extend from a class. This means that some types in haXe actually contain a hidden member variable within them that represent the Neko data value. The class you would normally use
within haXe to represent the values type is merely an object providing extended functionality for use in your haXe scripts.

For example, any value in haXe using the haXe String type is really an object instantiated from the haXe String class. You may recall a discussion about this in Chapter 3. Now, the String class does not fully scale down to a string represented within the Neko Virtual Machine, but the string value that the Neko Virtual Machine can see does exist within the haXe String object and can be extracted within an untyped environment by accessing the property __s. It is this property that you need to pass to the C/C++ function for processing, not the actual haXe String object. If you wanted to, you could extract this manually, as in the following code:

```java
var str = "Hello, World!"
hello( untyped str.__s );
```

Not all base types in haXe that extend a class require data extraction, as they still manage to scale correctly to the Neko Virtual Machine. However, it pays to stay on the safe side and subject all values passed to C/C++ with the neko.Lib.haxeToNeko function. The values specifically requiring restructuring are those of type String, and those of type Array.

### Restructuring Data in haXe from C/C++

Just as you have to prepare data to leave haXe for a C/C++ library, you are also required to restructure data returned from C/C++. This is done, surprisingly, using the neko.Lib.nekoToHaxe function. Again, this is not strictly necessary for all data types returned from the C/C++ code, though it often pays to be cautious and prepare all returned code. As with values passed to the C/C++ code, the values specifically requiring restructuring are those of type String, and those of type Array.

### Neko Type Checking in C/C++

One part of the HelloWorld example that has not yet been discussed is the line:

```java
val_check( str, string );
```

This line of code provides the type checking of the passed str parameter.

Working in an environment where all function parameters are not checked at compile time is a dangerous situation to find yourself in. The problem is that, should a value’s type not be what you expect, you could possibly crash the virtual machine altogether. This becomes especially precarious should your Neko Virtual Machine be functioning as an extension to web server software on a shared host machine, as any lack of safe data management will also affect other server users.

So, how do you type check a value type? Well, as one would expect, Neko provides a number of functions for just such a purpose. In fact, there is quite a considerable number, at least one for each Neko data type, plus a generic function for almost all values as detailed in the HelloWorld example.

Table 20-2 provides a complete list of the type checking functions and the types they verify.

Most of these functions should be pretty self-explanatory. Those functions that do require further explanation will be discussed later in this chapter.
Chapter 20: Extending haXe with C/C++

**Table 20-2**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val_check( value v,</td>
<td>Compares the C data type held in the value struct with the passed type</td>
</tr>
<tr>
<td>void* t )</td>
<td></td>
</tr>
<tr>
<td>val_is_null( value v )</td>
<td>Returns true if the value is null, or else false</td>
</tr>
<tr>
<td>val_is_int( value v )</td>
<td>Returns true if the value is an int type, or else false</td>
</tr>
<tr>
<td>val_is_float( value v</td>
<td>Returns true if the value is a float type, or else false</td>
</tr>
<tr>
<td>val_is_number( value v</td>
<td>Returns true if the value is a either an int type or a float type, or else</td>
</tr>
<tr>
<td>)</td>
<td>false</td>
</tr>
<tr>
<td>val_is_string( value v</td>
<td>Returns true if the value is a pointer of type char, or else false</td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>val_is_bool( value v</td>
<td>Returns true if the value is a bool type, or else false</td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>val_is_array( value v</td>
<td>Returns true if the value is an array of values, or else false</td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>val_is_object( value v</td>
<td>Returns true if the value is a Neko object type, or else false</td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>val_is_function( value</td>
<td>Returns true if the value is a function, or else false</td>
</tr>
<tr>
<td>v )</td>
<td></td>
</tr>
<tr>
<td>val_is_abstract( value</td>
<td>Returns true if the value is an abstract (miscellaneous) type, or else</td>
</tr>
<tr>
<td>v )</td>
<td>false</td>
</tr>
<tr>
<td>val_is_kind( value v,</td>
<td>Returns true if the value is an abstract (miscellaneous) type and has</td>
</tr>
<tr>
<td>void* t )</td>
<td>been flagged with a given identifier, or else false</td>
</tr>
</tbody>
</table>

The most common function you will use when type checking is `val_check`, as it allows for the broadest possible use while also providing the simplest of input. The type values it can accept for comparison are detailed in Table 20-3.

**Table 20-3**

<table>
<thead>
<tr>
<th>Type</th>
<th>haXe Mapping</th>
<th>C/C++ Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>untyped String.__s</td>
<td>Pointer to type char (char*)</td>
</tr>
<tr>
<td>int</td>
<td>Int</td>
<td>int</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
<td>float</td>
</tr>
<tr>
<td>number</td>
<td>Float or Int</td>
<td>float or int</td>
</tr>
<tr>
<td>bool</td>
<td>Bool</td>
<td>bool</td>
</tr>
<tr>
<td>array</td>
<td>Array</td>
<td>Pointer to type value (value*)</td>
</tr>
<tr>
<td>object</td>
<td>A class instance or Dynamic value</td>
<td>N/A; object properties can only be accessed with the use of C helper functions</td>
</tr>
</tbody>
</table>
Although \texttt{val\_check} appears to be the ultimate type checking function, this doesn’t mean it makes the other type checking functions redundant. While most of the functions in the previous table referring to the lost of type checking functions return a nice safe Boolean value, \texttt{val\_check} is a much harsher customer; if the check verifies as true, then nothing happens and nothing is returned meaning life can go on as normal, but if the opposite occurs, a \texttt{neko\_error} is thrown and the currently executed Neko script is exited. This might not be what you want, and certainly can be a little rash in some cases. At worse, it can appear rude to your library users who probably won’t have a clue as to what went wrong. Often, then, it might be preferred to compare the data type, and following an unsatisfactory result, exit the currently running function, while allowing the Neko script to continue its execution, or at the least, provide the user with a friendly comment.

Most often, the functions you write in your C/C++ libraries will be wrapped by haXe classes you write yourself. Under such circumstances, it might be safe to assume that any data type checking becomes redundant, as you will undoubtedly have safeguarded all data transfer to and from the library. However, it still pays to check the data, and deal with any unforeseen eventualities. You never know when a library user has taken it upon themselves to forgo your haXe classes in favor of streamlining their development.

**Passing C/C++ Values to Neko**

Okay, so you’ve seen how to prepare and pass values to your C/C++ library, but what about returning values back to the Neko script? If you’ve modified a value sent from the Neko script or you’ve created a new value in C/C++, you’ll need to package this into a \texttt{value} struct for the return trip. To do this, the Neko library provides a series of allocation functions. Table 20-4 lists these functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{alloc_int} (\text{int} \ v)</td>
<td>Returns a Neko value from a C/C++ int.</td>
</tr>
<tr>
<td>\texttt{alloc_float} (\text{float} \ t)</td>
<td>Returns a Neko value from a C/C++ float.</td>
</tr>
<tr>
<td>\texttt{alloc_bool} (\text{bool} \ b)</td>
<td>Returns a Neko value from a C/C++ bool.</td>
</tr>
<tr>
<td>\texttt{alloc_string} (\text{char} *\text{str})</td>
<td>Returns a copy of the string in a Neko value.</td>
</tr>
<tr>
<td>\texttt{alloc_empty_string} (\text{unsigned} \ \text{int} \ \text{size})</td>
<td>Returns a Neko value struct containing an empty string value that is a given number of bytes in size.</td>
</tr>
<tr>
<td>\texttt{copy_string} (\text{char} *\text{str}, \text{int} \ \text{val_size})</td>
<td>The same as \texttt{alloc_string()}, except it will only copy a given number of bytes.</td>
</tr>
<tr>
<td>\texttt{alloc_object} (\text{value} \ o)</td>
<td>Returns a C/C++ construct representing a Neko object within a value struct. The value struct can also be \texttt{NULL}.</td>
</tr>
<tr>
<td>\texttt{alloc_array} (\text{unsigned} \ \text{int} \ n)</td>
<td>Creates an array of type value (value*) with a given size.</td>
</tr>
<tr>
<td>\texttt{alloc_abstract} (v\text{kind} \ k, \text{void} *\text{data})</td>
<td>Creates a value from a pointer to abstract data and flags the value with the given identifier.</td>
</tr>
</tbody>
</table>
Chapter 20: Extending haXe with C/C++

Each of these functions leaves the original value intact, meaning copies of the data are made where necessary. To see the Hello World example returning a value, the following could be possible:

```c
#include "neko.h"
#include <stdio.h>
value PrintHelloWorld( value str )
{
    val_check( str, string );
    printf( val_string( str ) );
    return alloc_string( "Hello to you, too, Neko!*" );
}
DEFINE_PRIM( PrintHelloWorld, 1 );
```

**Abstract Types**

One of the most likely reasons for creating a Neko library will be to provide bindings for an existing library in C/C++ offering features that Neko doesn’t currently support. When dealing with such libraries, it will become likely that passing standard string or number type values between C/C++ and Neko will be insufficient for your needs. A probable reason for this is that a data type used by the library that requires persisting within Neko is not of a type that Neko will recognize.

**Allocating Abstract Values**

Be it a C struct or other such data, you can deal with these occurrences by wrapping a pointer to the value within a value struct and labeling the value as an abstract type.

```c
value myAbstract = alloc_abstract( MY_ABSTRACT_KIND, &someData );
```

The value `MY_ABSTRACT_KIND` is an identifier for the particular type of data you are wrapping. This is not needed by Neko, but is very important for future use within your C/C++ library. For example, imagine having several different types of abstract data you want wrapped and passed to the Neko layer. Without some way to distinguish each value, you could quite easily misuse the data and cause the virtual machine to crash, or worse.

Abstract type identifiers are known by the Neko language, and indeed its community, as *kinds*. You can call a kind anything you like, but you must be sure to declare the kind in every C/C++ file in which it is used. You declare a kind using the macro `DECLARE_KIND`. Normally, this will occur inside a header file and that file will then be included within each C/C++ file that the particular abstract value is used:

```c
DECLARE_KIND( MY_ABSTRACT_KIND );
```

`DECLARE_KIND` is only useful for signifying the type identifier’s existence to the C/C++ compiler. In order to actually use the identifier, it must first be defined. Surprisingly, you do this using the `DEFINE_KIND` macro:

```c
DEFINE_KIND( MY_ABSTRACT_KIND );
```

Unlike the `DECLARE_KIND`, `DEFINE_KIND` must only occur once within your C/C++ library, so it wouldn’t normally appear inside a header file.
Handling Abstract Values in Neko

Once you have your nicely packaged abstract value in C/C++, you’re probably going to want to pass it back to Neko. In haXe, an abstract value doesn’t have a type, per se, but as all values in haXe have a type, even if inferred, it is then important to tell the haXe compiler that the variable container holding the abstract value is of a nonmanageable type or placeholder if you will. This is especially true if the variable container exists as a class member variable, whereby failing to specify a type results in a compiler error. So, you make your value safe and type the variable as type Void:

```haXe
def var myAbstract : Void = hello( neko.Lib.haxeToNeko( "Hello, World!" ) );
```

This now has the benefit of being a valid type, which is not null, yet it is also not a value that can be processed as such, except to have its reference exchanged or duplicated. This proves perfect for using an abstract value, as the only purpose it will serve is to pass back to further C/C++ functions for processing.

Here is a simple, but complete, example of an abstract in use:

```c
define_kind( k_str_struct );

struct _strings {
    char *str1;
    char *str2;
};
typedef struct _strings strings;
value GetAbstract( value str )
{
    val_check( str, string );
    strings* strStruct = (strings*)malloc(sizeof(strings));
    strStruct->str1 = "Howdy";
    strStruct->str2 = val_string( str );
    return alloc_abstract( k_str_struct, strStruct );
}
value PrintAbstract( value abstract )
{
    val_check_kind( abstract, k_str_struct );
    strings* strStruct = (strings*)val_data( abstract );
    printf( "%s %s", strStruct->str1, strStruct->str2 );
    return val_null;
}
define_prim( GetAbstract, 1 );
define_prim( PrintAbstract, 1 );
```

And the haXe class to manage this library:

```haXe
class AbstractHandler
{
    static var __a : Void;
    public static function main()
    {
```

564
Chapter 20: Extending haXe with C/C++

```java
__a = getAbstract( neko.Lib.haxeToNeko("Doody") );
printAbstract( __a );
}
static var getAbstract = neko.Lib.load( "hello", "GetAbstract", 1 );
static var printAbstract = neko.Lib.load( "hello", "PrintAbstract", 1 );
```

When dereferencing an abstract value struct, you must remember that the function `val_data` returns the content of the struct as type `void *`, so the C/C++ compiler doesn’t really know what it’s dealing with. Therefore, you must cast the value referenced from the value struct before assigning it elsewhere.

### Object Types

haXe objects function in C/C++ much like key/value tables. They allow grouping of multiple values, including other objects and Neko functions, by name. This proves useful when passing numerous values of varying type from a C/C++ library to a Neko script as part of the return value from an executed C/C++ function. Each of the values within a Neko object can be of differing type and you are not restricted to the number of values assigned to the object.

### Passing Objects from Neko to C/C++

One of the coolest aspects of Neko objects is that they map directly to haXe class instances. When passing a class instance from the Neko layer to the C/C++ library, each of the member functions and variables from that class instance are then directly accessible by the C/C++ library. Likewise, any object constructed within the C/C++ layer can be cast into a haXe class instance when returned to Neko. This creates for incredibly flexible data management between the C/C++ library and the Neko Virtual Machine, and opens up a world of possibilities in terms of Neko to C/C++ interaction. The only issue one need remember is data structure preservation with regard to minimizing possible runtime errors.

Objects are a little less straightforward than dealing with individual value structs. Neko does provide the needed flexibility, as always, though the readability of your C/C++ code may suffer slightly through their use. The following is an example of an object being passed from Neko to a C/C++ library:

```java
class Object
{
    public var myInt : Int;
    public var myStr : String;
    public var myObj : Dynamic;
    public function new()
    {
        myInt = 78;
        myStr = "Neko rocks!";
        myObj = { one: 12, two: 22 };
    }
}
class ObjectHandler
{
    public static function main()
    {
        var obj = new Object();
        printObject( obj );
    }
    static var printObject = neko.Lib.load( "ObjectLib", "PrintObject", 1 );
}
```
Part III: Extending the Possibilities

And the C/C++ code:

```c
#include "neko.h"
#include <stdio.h>
value PrintObject( value obj )
{
value nestedObj;
int myInt;
char* myStr;
int one, two;
val_check( obj, object );
nestedObj = val_field( obj, val_id( "myObj" ) );
myInt = val_int( val_field( obj, val_id( "myInt" ) ) );
myStr = val_string( val_field( val_field( obj, val_id( "myStr" ) ),
val_id( "__s" ) ) );
one = val_int( val_field( nestedObj, val_id( "one" ) ) );
two = val_int( val_field( nestedObj, val_id( "two" ) ) );
printf( "%i, %s, %i, %i", myInt, myStr, one, two );
return val_null;
}
DEFINE_PRIM( PrintObject, 1 );
```

From this example, there is very little that should appear as new within the actual haXe code, but the C/C++ code should be a different kettle of fish. As you can probably make out, the fields of the passed haXe class instance are accessed using the `val_field` function. This function returns a `value` struct, so it needs to be de-referenced before its values can be used by C/C++ functions. The `val_field` function takes two arguments: the object containing the field to extract and the field identifier. As fields names are not stored as actual strings within an object table, but as hashed equivalents of those names, the field identifier needs to be looked up using the `val_id` function.

As noted earlier when preparing haXe code for transit to C/C++, the C readable string value stored in a haXe `String` object is contained in a field called `__s`. As a `String` object is like any other Neko object, the `__s` field needs to be extracted in the same manner as any object field in order to obtain the actual textual string `value` struct.

Once you have extracted the `value` struct stored in an object, you can pretty much do what you will with it. Any fields extracted from an object do not alter its content, except where the data held in a field is a pointer reference to data in memory.

**Creating and Modifying Neko Objects in C/C++**

As you would expect from the previous table showing Neko library allocation functions, you create a Neko object in C/C++ using the `alloc_object` function. Creating an object in this way without an existing object is merely a means to initialize a new `value` struct, readying it for the allocation of values via newly assigned fields:

```c
value myObj = alloc_object( NULL );
```

Passing an existing object to `alloc_object` copies all values from the original object to the new object. Any values containing pointers to data will have the pointer reference copied, not the data the pointer points to.
Once you have your new object structure, or perhaps an existing object, you can then assign value structs to it using the `alloc_field` function. `alloc_field` accepts three parameters. The first two are identical to those required by `val_field`, while the third is the value struct you wish to assign to the object:

```c
alloc_field( myObj, val_id( "newField" ), myValue );
```

You can add as many fields to an object as you wish, and are not limited in any way, except to say that any added fields must be of the value struct type.

### Arrays

Like many array types in the C/C++ languages, Neko Arrays are presented in the C/C++ layer as a pointer to a data type, in this case a value struct, or `value*`. In order to get the value pointer from a `value` object, you first need to pass the `value` to the `val_array_ptr` function, which will return the pointer for you. From there, it is simply a case of extracting the values contained therein with the help of the `val_array_size` function for the upper bounds limit reference. For example, suppose an array of an unknown number of integer values were sent to the C/C++ library, the code to extract and print the data could look something like this:

```c
value* p_arr;
int i;
if ( val_is_array( intArray ) )
{
    p_arr = val_array_ptr( intArray );
    for ( i = 0; i < val_array_size( intArray ); i++ )
    {
        if ( val_is_int( p_arr[i] ) )
            printf( "item %i = %i", i, val_int( p_arr[i] ) );
        else
            printf( "non-integer value found at item %i", i );
    }
}
else
    printf( "value is not an array" );
```

Neko Arrays, or pointers to value structs, are subjected to the typical rules for pointers to item lists in C/C++. One of the biggest limitations is that, although you can increase the size of an Array in Neko exponentially, without requiring any formal re-dimensioning, you are pretty much capped to an initial limit when using the same Array in the C/C++ layer.

When creating an Array to return to Neko, you use the `alloc_array` function, which accepts an integer value depicting the number of items you can store in the Array. Once set, this cannot be changed, so any re-dimensioning requirements will force you to copy the Arrays stored values into a new Array set to the appropriate size.
Part III: Extending the Possibilities

Remember that a Neko Array can only contain value items. You probably think this sounds obvious, but when dealing with data in C/C++, you may forget to wrap data into a value before storing in an Array, which can lead to hair pulling until you realize the simple mistake.

As with haXe strings, the haXe Array also requires that the actual array data be extracted from a field of the Array object before using it in C/C++. In this instance, the field containing the array data is called __a. To extract the data, simply perform the process of extracting data from the field __a of the object Array.

Functions

As with anything else, pointers to functions can also exist as value types in the C/C++ layer. Passing functions to and from C/C++ is almost criminally simple, and only boosts the power and simplicity of Neko on the whole. Neko functions can be applied to Neko objects as fields, stored in an Array, or passed individually. Once acquired in C/C++, the Neko API provides a series of helper functions to aid in querying a function’s requirements and to actually make calls to the function.

Passing Functions from haXe to C/C++

Functions in Neko are passed to C/C++ by reference, much like objects. The structure of the function is unchanged when passed to and from C/C++, as the passed data can be thought of as being merely C code stored within a value struct, much like everything else. This means that, just as you can pass a C integer into and out of a Neko script, so can you pass a C function into and out of a Neko script. Both C and Neko functions are interchangeable and can be called from practically anywhere.

To pass a function to the C/C++ layer is as simple as passing an integer value, too. You can even construct the function as the parameter to be sent to the C/C++ layer rather than first assigning the function reference to a variable, though it would only be recommended if the function were small, for readability sake. For example, here is a simple print replacement function written in Neko to be called from C/C++:

```neco
class FunctionPass
{
    public static function main()
    {
        printReplacement( function( str : String ) { neko.Lib.print( str ); } ) ;
    }
}
```

It is the same as writing:

```neco
class FunctionPass
{
    public static function main()
    {
        var func = function( str : String )
        {
            neko.Lib.print( str );
        }
        printReplacement( func ) ;
    }
}
```

568
When coding in haXe, you can choose to pass either class member functions or local functions, as both should function in the same way. However, it sometimes is found that when passing a class member function, the reference to the function can get lost, which can cause an application to fail; therefore it would strongly be recommend using local functions where possible.

**Passing Functions from C/C++ to haXe**

As with all types in the C/C++ layer that are recognized by Neko, you package functions with an alloc helper function, in this case — you guessed it — with alloc_function.

```
value val_func = alloc_function( pFunc, intParams, strName );
```

*alloc_function* accepts three parameters: The first is the C function pointer you wish to send to Neko, the second is the number of parameters the function accepts, and the third is a string value depicting the name of the function, for use by Neko. As the third parameter will be used as the function name within Neko, it must follow the same rules for Neko function naming conventions.

When the function is retrieved by Neko, you can store it into a variable, ready to call when needed. At this point, there is no error checking on the data types passed to the function, so it pays to do this when typing the variable containing the function reference. For example:

```
var cFunction : Int -> String -> Void = getCFunc();
```

If you’re writing the haXe classes and C/C++ library for others to use, it should fall as your responsibility to see that users of the library do not make needless mistakes.

**Executing Neko Functions in C/C++**

There is no way to extract a function pointer from a *value* struct, so to call a Neko function, or a function stored into a *value* struct by C/C++, you will need to make use of one of a series of helper functions. There are quite a few of them, to tell the truth, but you really only need to use one of them:

```
value ret = val_callEx( val_this, val_func, val_args, intNumArgs, &val_exc );
```

It looks a little complicated, huh? This is why there are so many helper functions to call Neko functions. This particular one is the generic helper, while the others make life simpler providing certain conditions are met.

For *val_callEx*, the parameters required are as follows:

- **val_this**: This should be a *value* struct containing a Neko object that you want to set as the context for the particular executing function.
- **val_func**: This is the *value* struct containing the Neko function to call.
- **val_args**: This is a Neko array containing the *value* structs required as parameters for the function call.
- **intNumArgs**: This is the number of parameters stored in *val_args*.
- **val_exc**: This is a pointer to a *value* struct you want to use to catch any exceptions that occur. Passing **NULL** will allow the exception to filter into the C/C++ layer for handling after the function call.
Part III: Extending the Possibilities

Most of the time, this function will probably be a little overkill. You can, however, choose an easier, more readable route, thanks to a series of functions, each one oriented to the number of parameters required by the function you want to call. These are `val_call0`, `val_call1`, `val_call2`, `val_call3`, and `val_callN`. `val_callN` is very much the same as `val_callEx`, only without the exception and this parameters. This can work well, as you will likely very rarely require the use of these two parameters:

```
val_callN( val_func, &val_args, intNumArgs );
```

The other functions accept the number of parameters as specified by their name, such as:

```
val_call0( val_func )
val_call1( val_func, val_arg );
val_call2( val_func, val_arg1, val_arg2 );
val_call3( val_func, val_arg1, val_arg2, val_arg3 );
```

All of these functions will return a `value` struct if successful, even if that value is a `val_null`.

Managing Neko Primitives

Much of dealing with building C/C++ libraries for Neko involves the many data type conversions, which you have already learned earlier in this chapter. The helper functions provided greatly reduce the pain of managing data and exposure of this data between the C/C++ and the Neko layers. However, there are limitations to these helper functions.

One of the primary issues with what you have learned so far revolves around the number of parameters you can safely assign to the signature of a C/C++ function destined to be exposed to the Neko layer. The problem is, for every possibility of function signature, there is a corresponding helper capability. Therefore, code has been written for functions accepting one parameter, those accepting two parameters, those accepting three, and so on. This goes on to functions accepting up to five parameters. From there, if you wish to expose a function in your library utilizing more than the maximum five parameters, you’ll need to handle the parameter data differently.

For example, normally, for a function accepting a few parameters, you may detail its signature as:

```
value someFunc( value one, value two, value three )
```

Once written, you will then expose this function using the `DEFINE_PRIM` function, like this:

```
DEFINE_PRIM( someFunc, 3 );
```

Great. This should work glitch free, providing the rest of your code has no bugs. Now, suppose you wish to add a further three parameters to this function. If the `DEFINE_PRIM` helper function only caters for functions accepting up to five parameters, how do you expose a function requiring more? The answer is in a new helper function, `DEFINE_PRIM_MULT`.

```
DEFINE_PRIM_MULT( someFunc )
```

`DEFINE_PRIM_MULT` works in a similar fashion to `DEFINE_PRIM`, with the exception that it doesn’t require a value depicting the number of parameters required by the passed function. To facilitate this,
any function written to accept more than five values when called needs to utilize a different style of signature. Instead of allowing an arbitrary number of parameters, the function should instead accept a guaranteed two parameters: a pointer of type value, or value*, and a C/C++ integer. Here, the first parameter represents an Array of values, and should be treated as such, while the second parameter contains the number of values stored in the Array. This now gives you the flexibility to access an almost limitless number of parameters while maintaining simplicity:

```
void someFunc( value* args, int nargs )
```

When declaring a C/C++ function, in haXe, that accepts more than five parameters, you must set the number of parameters value to -1 in the neko.Lib.load function call. This way, the Neko layer does no checking on the number of values being sent to the C/C++ function.

**Garbage Collection**

The term garbage collection is enough to make most developers cringe. Memory management is always a tough challenge for even the hardiest, most seasoned developer, but unfortunately it has to be dealt with. From the user’s point of view, an application that consumes memory like a hungry beast without relinquishing it is very frustrating, and darn right rude.

Luckily, Neko uses the excellent libgc garbage collection library, and to good effect. Of all the existing functionality within Neko, you’d be hard pressed to get it to fall over because of memory leaks. However, if you’re extending Neko using C/C++, you fall into a realm where Neko has limited control, so it is still down to you, the developer, to manage where memory is being used, and to ensure that it is cleaned up properly.

When storing data into a value struct for persistence, you’ll probably also want to request support from the garbage collector for that value, so that, once the value is no longer required, it can be freed safely and efficiently. Neko provides several functions for just such a purpose.

**alloc_root()**

When working with event-based libraries, you will more than likely acquire the need to deal with some form of callback to the Neko layer. This callback will fire when a particular event is raised, so, you never know when the function will be called, only that you need to register a function to the event in advance.

Neko provides a solid way of handling this kind of scenario, whereby a Neko function can be stored into a static variable, allowing a C/C++ event handler function to act as a proxy, thereby calling the stored Neko function. This Neko function is then managed by the Neko garbage collector, so it is only fully released sometime after a request to free the data has been made and, thus remains at your disposal for as long as you need it.

The storage medium is a pointer to a value struct, or value*. As this is static, you’ll need to declare this outside of any C/C++ function scope:

```
value* function_callback = NULL;
```
Part III: Extending the Possibilities

When you assign a function value to this store, you will need to allocate memory to it using the `alloc_root` Neko helper function. This also ensures that it is registered with the Neko garbage collector, making the function safe:

```c
value set_callback( value f ) {
    val_check_function(f,1); // checks that f has 1 argument
    if( function_callback == NULL )
        function_callback = alloc_root(1); // allocate memory
    *function_callback = f; // assign the function to the store's address
}
```

As the static function store variable is a pointer, you can essentially store multiple functions using this pointer as you would with an Array. However, like Arrays, you have to define how many items will be stored with this pointer, even if it is just one value. This is where `alloc_root` becomes useful. Not only does it register the pointer address with the garbage collector, but it also allows the setting of the number of value structs stored using the pointer, represented by its solitary parameter.

**free_root()**

`free_root` is the second half of `alloc_root`, in that it frees any memory allocated with `alloc_root`. This is not done immediately, however, but at the leisure of the garbage collector, which promises to tidy the memory sometime after the call is made, but no sooner.

It is important to remember that the garbage collector will not free memory allocated with `alloc_root` without the explicit call to `free_root`.

**alloc()**

`alloc` is a function very similar in use as the C function `malloc`. It is used for values that are not static, thus can and should be used where `alloc_root` cannot.

Unlike `alloc_root`, `alloc` doesn’t need to be explicitly freed. This is because any memory assigned by `alloc` will be freed by the garbage collector sometime after the last pointer to reference the allocated memory is no longer reachable.

Like `malloc`, `alloc` accepts a single parameter stating the number of bytes you want stored in the newly garbage collector–managed memory. If you don’t know the size of bytes required to, say, store an object in the allocated memory, then use the C function `sizeof` to acquire it:

```c
SomeStruct* data = alloc( sizeof( OtherObject ) );
```

**Garbage Collecting Abstract Values**

If you’re dealing with Neko values, then you don’t need to worry about garbage collection, as Neko already handles this. However, any abstract values you use in your libraries are kind of an unknown entity, so Neko chooses not to handle them. Luckily, though, Neko does provide a way to callback a set function when garbage collection should occur on an abstract value, so that you can handle this yourself.
To flag an abstract value for a garbage collection callback, you need to use the function `val_gc`. This function takes two parameters: The first parameter is the `value` struct you wish to mark for the callback, while the second parameter is a function pointer to the function that will handle the custom garbage collection. Here is an example of how this might work:

```c
void dispose( value v ) {
    cleanup( val_data(v) );
}
value create() {
    void *ptr = ....
    value v = alloc_abstract(k_somekind,ptr);
    val_gc(v,dispose);
    return v;
}
```

Now, at the time when the garbage collector is set in motion, which is often the time when the Neko script ends, the callback function is fired, thus activating your own cleanup code.

## Error Handling

Handling errors is an important factor of any programming, but with regard to C/C++ libraries extending the Neko Virtual Machine, where should they be handled? For the most part, a return value for the offending code can easily be useful to determine whether a function was successful or not, but often you’ll want to be a bit more extreme should the fault be of a serious nature.

### `val_throw()` and `val_rethrow()`

The `throw` and `rethrow` functions are great for handling errors in code that you feel should be handled by the Neko script calling the C/C++ library. Both functions are very similar, accepting a `value` struct as its solitary parameter to be used as the exception object. The difference being, though, that while `throw` deals with the creation of an exception, `rethrow` will forward an existing exception that otherwise would have been handled.

```c
val_throw( alloc_string( "An unknown error occurred" ) );
```

Exceptions thrown by the C/C++ layer should be handled in your Neko scripts. For more information on exceptions, see Chapter 7, “When Things Go Wrong.”

### `failure()`

The `failure` function is very useful when creating your own libraries, as it allows you to pass a string constant for display in the command prompt, while raising a Neko error. This is great when you quickly want to debug something without having to resort to large amounts of code in haXe, or if the error found in the C/C++ code is something that should not be recovered from. The following is a simple example of how `failure` might be used:

```c
if ( error != NULL ) failure( "Several errors have occurred" );
```
Part III: Extending the Possibilities

This example would halt the Neko script, while printing *Several errors have occurred* in the command console.

**bfailure()**

The `bfailure` function performs the same feat as `failure`, with the exception that, rather than a string constant, it accepts a Neko buffer value.

Neko buffers are created with `alloc_buffer`, which as you might expect, allocates enough memory for a buffer and registers the structure with the garbage collector. Once created, you can pass a Neko string value to the buffer using `val_buffer` or a constant string — a value of type `char*` — using `buffer_append`. At any time, you can convert the buffer to a Neko string using `buffer_to_string`:

```plaintext
value str1 = alloc_string( "2 + 2" );
value str2 = alloc_string( "4" );
buffer buf = alloc_buffer( "The sum " );
val_buffer( buf, str1 );
buffer_append( buf, " = " );
val_buffer( buf, str2 );
return buffer_to_string( buf );
```

**val_print()**

`val_print` is a very useful function that could save you large amounts of hair pulling and certainly lots of time wasted writing needless code. When receiving data into a function written in C/C++ from a Neko script, you will often just want to know what is stored at a glance within the `value` struct container. Now, usually, in order to examine its content, you would have to extract the data contained in the `value` and somehow convert it to a format that you can output. This can certainly be a pain in the rear if the value type is complex, such as a Neko object. Thankfully, Neko provides the `val_print` function, which accepts a `value` struct as a parameter, and writes the content of the `value` struct to the command console as a string.

**neko_error()**

`neko_error` doesn’t currently do a lot. Its use is currently exactly the same as returning the C `NULL` value from a function, which inevitably raises a Neko exception. However, this may be subject to change in the future, so its use is preferred over the explicit use of `NULL` in a return value.

If you wish to return the Neko equivalent of `NULL`, use `val_null`.

**Summary**

This chapter is more of a reference than a learning aid. For the seasoned C/C++ developer, hopefully you managed to glean a lot of how the Neko C foreign function interface could be of use in extending the haXe language to suit your application development needs. For those who aren’t competent with either C or C++, however, it is urged that you pick up a good book on one of these languages and start.
learning. There is little in haXe and Neko, or indeed any other language that is more satisfying than creating extensions with your own libraries. Combined with the simplicity of the Neko C FFI, you have everything at your disposal to create any application imaginable.

In this chapter, you learned:

- How to use the Neko C FFI conversion and allocation functions
- How to create your own Neko ndll library extension
- How to explicitly use the garbage collector
- How to throw errors or trace values in the C/C++ layer
- How to pass functions and values between the C/C++ layer and Neko
This appendix describes the most used and useful HTML 4.0 elements and their attributes. All the definitions that only have formatting purposes, or that have been marked as obsolete from the older versions, have been discarded. All the formatting of the HTML document can be done using properly written CSS definitions.

The following sections describe groups of semantically related elements. Following the textual description, a table defining each element individually is provided. Beyond the element name and the description, the second column indicates if the element is of the empty type (it cannot contain other nodes) and the third column contains the names of the attribute groups or attributes that apply to the element. When you want to know more about an attribute, look for its name first in the table for attribute groups, and then in the table for attribute names in case you do not find it.

## Document Definition Elements

This section contains the elements that are required to define an HTML document properly. The meta element adds optional information on the document as a whole.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Empty?</th>
<th>Frequently used Attributes or Attribute Groups</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>body</td>
<td>false</td>
<td>onload, onunload, i18n, core, events</td>
<td>Defines the main content area of the document. It must be a child of the html element and must be unique in the document.</td>
</tr>
</tbody>
</table>

*Table continued on following page*
Appendix A: Semantic HTML

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Empty?</th>
<th>Frequently used Attributes or Attribute Groups</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>head</td>
<td>false</td>
<td>i18n, profile</td>
<td>Defines the heading section of a document. It contains information that does not have a direct visual representation. In any document just one head element contained in the html element must exist.</td>
</tr>
<tr>
<td>html</td>
<td>false</td>
<td>i18n</td>
<td>Is the root element of every (X)HTML document. Any document must contain just one html element.</td>
</tr>
<tr>
<td>meta</td>
<td>true</td>
<td>http-equiv, name, content, scheme, i18n</td>
<td>Defines a piece of information that may be used for different purposes by the client. For example, a search engine may use the meta information to better index the content of the document.</td>
</tr>
<tr>
<td>title</td>
<td>false</td>
<td>i18n</td>
<td>Defines a short description of the document as a whole. It must exist in the head element.</td>
</tr>
</tbody>
</table>

**Structural Elements**

Structural elements are used to define sections in a document. They exist in two forms: block and inline. In the first case the div element is used; a common application for div is, for example, to distinguish the header section of a page, the main content area, and the footer section as illustrated in the following example:

```html
<html>
  <head>
    <title>Sample Page</title>
  </head>
  <body>
    <div id="header">
      Header of the page goes here
    </div>
    <div id="main">
      Content of the page
    </div>
    <div id="footer">
      Footer goes here
    </div>
  </body>
</html>
```
Appendix A: Semantic HTML

Note the use of the `id` attributes to label the sections. In the same page, no more than one element with the same element `id` can exist. Each `div` will contain more elements like paragraphs, tables, sub-blocks, and others.

The `span` element is used in the same way but to define sections inside a line of text. It is possible to mark individual words with a `class` or an `id` attribute.

```html
<p>The item price is <span class="cost">$ 100.00</span>.</p>
```

In the preceding example, the currency value is marked with a `class` attribute with value cost. A `class` has been used because the cost value can be used more than once to mark various elements in the same document. An important consideration is that block elements can contain other block elements or inline elements, whereas inline elements can contain only other inline elements.

The `div` and `span` elements should be used to add semantic value to a text. Quite often those elements are used with the sole purpose of marking the document for formatting reasons; this practice must be avoided as much as possible: The semantic structure is almost always good enough to define suitable areas to be formatted and to apply visual effects to.

The `div` and `span` elements are not the only way to define block and inline sections but they are dealt separately because they do not provide any additional semantic. A `p` element is a block-type element with the additional semantic of representing a paragraph. The `em` element is an inline element with the additional semantic of being an emphasis.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Empty?</th>
<th>Frequently used Attributes or Attribute Groups</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>div</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Defines a block section. A block may be a group of paragraphs mixed with tables, images, and so on.</td>
</tr>
<tr>
<td>span</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Defines an inline section that is a portion of text inside a phrase.</td>
</tr>
</tbody>
</table>

They are the most basic and fundamental elements in an HTML document. The more frequently used elements of this group are `p`, `br`, and the `h1` to `h6` tags. The first wraps a paragraph, the second adds a line break (without interrupting the paragraph continuity), and the last are used to define titles and subtitles.

```html
<h1>Main title</h1>
<p>A small text introduction.</p>
<h2>A subtitle</h2>
<p>More text goes here.</p>
<p>...</p>
```
<table>
<thead>
<tr>
<th>Element Name</th>
<th>Empty?</th>
<th>Frequently used Attributes or Attribute Groups</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbr</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Is used to enclose an abbreviation. The extended version is usually located in the title attribute.</td>
</tr>
<tr>
<td>acronym</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Is the same as abbr but to mark an acronym. <code>&lt;acronym title=&quot;Hypertext Markup Language&quot;&gt;HTML&lt;/acronym&gt;</code></td>
</tr>
<tr>
<td>blockquote</td>
<td>false</td>
<td>cite, i18n, core, events</td>
<td>Wraps a quotation. It is used for long quotations (block content); for short quotations see q. The cite attribute is used to point a resource that contains additional information about the quotation.</td>
</tr>
<tr>
<td>br</td>
<td>true</td>
<td>core</td>
<td>Inserts a line break in the text.</td>
</tr>
<tr>
<td>cite</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Is used to enclose a citation.</td>
</tr>
<tr>
<td>code</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Designates a fragment of computer code.</td>
</tr>
<tr>
<td>del</td>
<td>false</td>
<td>cite, datetime, i18n, core, events</td>
<td>The marked portion of text has been removed from the document after its publication. The element is used in a revision context. The cite attribute can contain a URI to a document that justifies the alteration of the text while the datetime attribute is used to indicate the date and time when the edit has occurred.</td>
</tr>
<tr>
<td>dfn</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Wraps a term inside a phrase that defines its meaning.</td>
</tr>
<tr>
<td>em</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Inline element to add emphasis to words in a phrase. Usually it is rendered as italic text.</td>
</tr>
<tr>
<td>h1 to h6</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Heading elements define titles and subtitles. Lower numbers have greater priority: h1 is more important than h2 and others.</td>
</tr>
<tr>
<td>hr</td>
<td>true</td>
<td>i18n, core, events</td>
<td>Inserts a horizontal rule.</td>
</tr>
<tr>
<td>ins</td>
<td>false</td>
<td>cite, datetime, i18n, core, events</td>
<td>Denotes a portion of text that has been added in a later edit of the document. It is used for revision purposes along with the del element.</td>
</tr>
<tr>
<td>kbd</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Is used to mark a keyboard letter in a text that describes its usage.</td>
</tr>
<tr>
<td>p</td>
<td>false</td>
<td>i18n, core, events</td>
<td>A block element to delimit a text paragraph.</td>
</tr>
</tbody>
</table>
Appendix A: Semantic HTML

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Empty?</th>
<th>Frequently used Attributes or Attribute Groups</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Defines a section of preformatted text. The browser usually renders the text inside this element in a monospace font and maintains the whitespace semantic preserving multiple spaces and new lines.</td>
</tr>
<tr>
<td>q</td>
<td>false</td>
<td>cite, i18n, core, events</td>
<td>Wraps a quotation. It is used for short quotations (inline content); for long quotations see blockquote. The cite attribute is used to point a resource that contains additional information about the quotation.</td>
</tr>
<tr>
<td>samp</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Designates sample output from computer programs, scripts, and so on.</td>
</tr>
<tr>
<td>strong</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Similar to em, it adds emphasis to the text. Traditionally the element is rendered with boldface characters.</td>
</tr>
<tr>
<td>sub</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Is an inline element to mark subscripts. It is usually rendered in a smaller font and slightly below the containing line of text.</td>
</tr>
<tr>
<td>sup</td>
<td>false</td>
<td>i18n, core, events</td>
<td>As sub but for superscripts.</td>
</tr>
<tr>
<td>var</td>
<td>false</td>
<td>i18n, core, events</td>
<td>An inline element to define variable names used in a describing text.</td>
</tr>
</tbody>
</table>

Lists

This section contains the definition of elements that describe lists of values. The list type can be unordered, when the sequence is not really important, or ordered, when list elements are prefixed with an automatically generated symbol (a number or a letter). There is also a special type of list, the data list, that is a combination of definition terms (dt) and definition data (dd).

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Empty?</th>
<th>Frequently used Attributes or Attribute Groups</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dd</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Is an element used to encapsulate the data describing the previous data-term element.</td>
</tr>
<tr>
<td>dl</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Defines a list of glossary types composed by definition terms (dt) and definition data (dd).</td>
</tr>
<tr>
<td>dt</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Defines a term in a definition list.</td>
</tr>
</tbody>
</table>

Table continued on following page
Appendix A: Semantic HTML

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Empty?</th>
<th>Frequently used Attributes or Attribute Groups</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>li</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Is an element of a list used inside the ol or ul elements.</td>
</tr>
<tr>
<td>ol</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Defines an ordered list of elements. The elements are prefixed by an auto-incremented numeric or alphabetic symbol.</td>
</tr>
<tr>
<td>ul</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Defines an unordered list: a list of elements with no particular priority. Every element of the list is rendered with the same bullet symbol or no symbol at all if so defined in the style sheet rules.</td>
</tr>
</tbody>
</table>

Tables

The elements of this section are used to define tabular contents. A table element may contain rows as immediate children or inside table groups that are contained in the table; the table element supports multiple groups for headers, footers, and bodies. Each row can contain content cells or header cells. Columns can be grouped semantically in two ways: putting them inside a colgroup element or using one or more col elements to share common attributes.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Empty?</th>
<th>Frequently used Attributes or Attribute Groups</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>caption</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Defines the caption of the table that contains it. It is usually rendered by the browsers above the table contents.</td>
</tr>
<tr>
<td>col</td>
<td>true</td>
<td>span, width, cellalign, i18n, core, event</td>
<td>Is used to associate the same attributes to a set of columns. To group columns structurally use the colgroup element. The col element may only appear inside a colgroup element.</td>
</tr>
<tr>
<td>colgroup</td>
<td>false</td>
<td>span, width, cellalign, i18n, core, events</td>
<td>Is used to group a set of columns. The span attribute is used to define how many columns will be included in the group. The col elements inside the element represents one or more columns in the group. The colgroup element may only appear inside a table element.</td>
</tr>
<tr>
<td>table</td>
<td>false</td>
<td>table, i18n, core, events</td>
<td>Defines a table, a grid of rows, and columns.</td>
</tr>
</tbody>
</table>
Appendix A: Semantic HTML

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Empty?</th>
<th>Frequently used Attributes or Attribute Groups</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tbody</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Groups a set of rows containing values (not headings). One or more tbody sections may appear in the same table. The tbody element may only appear inside a table element.</td>
</tr>
<tr>
<td>td</td>
<td>false</td>
<td>cell, cellalign, i18n, core, events</td>
<td>Defines a cell containing data inside a table. The td element may only appear inside a tr element.</td>
</tr>
<tr>
<td>tfoot</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Works the same way as tbody but it is used to group footer rows. The tfoot element may only appear inside a table element.</td>
</tr>
<tr>
<td>th</td>
<td>false</td>
<td>cell, cellalign, i18n, core, events</td>
<td>Defines a cell containing a table data header. The th element may only appear inside a tr element.</td>
</tr>
<tr>
<td>thead</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Works the same way as tbody but it is used to group heading rows. The thead element may only appear inside a table element.</td>
</tr>
<tr>
<td>tr</td>
<td>false</td>
<td>cellalign, i18n, core, events</td>
<td>Wraps one or more cells (td or th) in a single table row. The tr element may only appear inside one of the following elements: table, tbody, thead, tfoot.</td>
</tr>
</tbody>
</table>

Hyperlinks, Images, and Objects

The elements described here allow you to refer to resources that are external to the document. Images and objects are embedded in the page visualization as part of the document itself.

A complete image inclusion can be defined as follows:

```html
<img src="http://www.example.com/images/logo.png" alt="Logotype of the example.com website. The first letter 'E' is boldface-black and uppercase, the other letters are small and grey" />
```

Links (a) and maps permit creating interactive areas to pass from the current document to other resources.

A typical hyperlink may have the following format:

```html
<a href="http://www.example.com/it/" title="The Example.com homepage for the Italian readers" hreflang="it-IT">Example.com</a>
```
Appendix A: Semantic HTML

Maps are sets of geometric areas overlaid to images. Each area may be associated to a distinct action. They can be of two types: server side or client side. When the first type is used, the coordinates relative to the map image are passed with the request and interpreted by the server. The client side already has a set of area defined and does not need a call to the server to choose the action to perform.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Empty?</th>
<th>Frequently used Attributes or Attribute Groups</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>false</td>
<td>hyperlink, i18n, core, events</td>
<td>The common use for this element is to create a link to another document. The URI of the pointed document must be set in the href attribute. Another use for the element recurs on the attribute name and it is used to create anchors inside a document. In this way it is possible for other documents to point to a fixed spot in the page. To link an anchor, append to the URI the cardinal symbol # followed by the name of the anchor.</td>
</tr>
<tr>
<td>area</td>
<td>true</td>
<td>shape, coords, href, nohref, alt, tabindex, accesskey, onfocus, onblur, i18n, core, events</td>
<td>Defines a geometric area inside a map definition. Each area may be associated to a different action, a document navigation, or a script execution.</td>
</tr>
<tr>
<td>img</td>
<td>true</td>
<td>image, tabindex, i18n, core, events</td>
<td>Embeds an image resource inside the document. The image is referenced using the src attribute.</td>
</tr>
<tr>
<td>map</td>
<td>false</td>
<td>name, i18n, core, events</td>
<td>A map element defines different active areas on an image element.</td>
</tr>
<tr>
<td>object</td>
<td>false</td>
<td>object, tabindex, i18n, core, events</td>
<td>Is used to embed an external object (like a flash movie or a java applet) inside the page. The types of supported objects varies a lot with the used browsers; usually the object type is handled by an external plug-in. The object element can only contain param elements.</td>
</tr>
<tr>
<td>param</td>
<td>true</td>
<td>id, name, param</td>
<td>One or more param elements may be included in the object element to provide additional information to the engine that manages the object to embed. The attribute values vary greatly from type to type of the included object.</td>
</tr>
</tbody>
</table>
Forms and Controls

When a site is not just a series of documents to read but it is an interactive tool that the user can use to manage contents (web mail applications, forums, wikis, or CMS), a way to edit those contents and submit them to a server is necessary. This is the purpose of a form that collects a group of control elements, one for each variable that the server requires. The set of controls is quite wide and includes buttons, text fields, selection controls, and other controls. Advanced controls such as date selection tools, sliders, or validation controls are nowadays produced by JavaScript developers mixing the base controls with client-side scripts. In the near future, it is probable that modern browsers will include more advanced controls to supply to this lack as defined in new standards like HTML 5 and XForms.

A very simple form containing a single text control and submit button, may have the following aspect:

```html
<form action="save.n" method="post">
  <input type="text" name="name" value="John" />
  <input type="submit" name="submit" value="Send Name" />
</form>
```

The action attribute says which server page must receive the content of the form (if absent the current URI will be used).

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Empty?</th>
<th>Frequently used Attributes or Attribute Groups</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>button</td>
<td>false</td>
<td>name, value, type, disabled, tabindex, accesskey, onfocus, onblur, i18n, core, events</td>
<td>It is functionally equivalent to an input element of type button but has richer rendering capabilities. The button element may have content like an image or extended text.</td>
</tr>
<tr>
<td>fieldset</td>
<td>false</td>
<td>i18n, core, events</td>
<td>Encapsulates a group of related controls and labels.</td>
</tr>
<tr>
<td>form</td>
<td>false</td>
<td>action, method, enctype, accept-charset, accept, name, i18n, core, events</td>
<td>Acts like a container for a set of controls. More than one form element can be present inside the body of a document. The form element also specifies to which URI the form content will be submitted, using the action attribute.</td>
</tr>
<tr>
<td>input</td>
<td>true</td>
<td>type, name, value, checked, disabled, readonly, size, maxlength, src, alt, usemap, ismap, tabindex, accesskey, onfocus, onblur, onselect, onchange, accept, i18n, core, events</td>
<td>Defines a user control. The control kind is defined by the type attribute. Some attributes have a meaning only if used in conjunction with the correct control type.</td>
</tr>
</tbody>
</table>

Table continued on following page
Appendix A: Semantic HTML

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Empty?</th>
<th>Frequently used Attributes or Attribute Groups</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>false</td>
<td>for, accesskey, onfocus, onblur, i18n, core, events</td>
<td>A label element is a short description associate to a control. The control must be included in the label element or otherwise referenced using the for attribute.</td>
</tr>
<tr>
<td>legend</td>
<td>false</td>
<td>accesskey, i18n, core, events</td>
<td>Allows adding a caption to a fieldset.</td>
</tr>
<tr>
<td>optgroup</td>
<td>false</td>
<td>disabled, label, i18n, core, events</td>
<td>Encapsulates more options in a thematic group.</td>
</tr>
<tr>
<td>option</td>
<td>false</td>
<td>selected, disabled, label, value, i18n, core, events</td>
<td>Represents a value in a list of possibilities. It must be contained in an optgroup element or in a select element. If the attribute value is not present, the content of the option element will be used as value on form submission.</td>
</tr>
<tr>
<td>select</td>
<td>false</td>
<td>name, size, multiple, disabled, tabindex, onfocus, onblur, onchange, i18n, core, events</td>
<td>Is a control that permits to choose one or more values, if the attribute multiple is present, from a list of possible values. Each allowed value is contained in an option element inside the select element. The select element must contain at least one option element.</td>
</tr>
<tr>
<td>textarea</td>
<td>false</td>
<td>name, rows, cols, disabled, readonly, tabindex, accesskey, onfocus, onblur, onselect, onchange</td>
<td>A multiline text control.</td>
</tr>
</tbody>
</table>

Style, Script, and Links

The original HTML syntax that contained a very small subset of elements has been growing considerably over the years. New elements, such as marquee and blink, have come out, sometimes introduced by browser vendors and not defined in any common standard. Many of them had the only purpose of formatting the document content in a more palatable way. Those elements have added complexity to the documents and this complexity has rendered the work of web designers really harder when the time of updates occurred. To solve the problem, the concept of style sheets has been introduced. The embedded or linked style sheets contain rules that describe the document from a formatting point of view, moving away from the text structure the style information. Elements such as font have been deprecated because they do not add any semantic value to the document. The style sheet rules are collected in the standard Cascading Style Sheet (CSS) definition as later described.
Describing the CSS syntax is out of the scope of this book. It is enough to say that a rule is a set of instructions that may be applied to elements. To decide which elements must be affected, one or more selectors are indicated in a comma separated list.

A CSS rule will have the following form:

```css
a {
  font-weight: bold;
}
```

Where `a` is the selector (the rule applies to all `a` elements) and inside the brackets are the formatting instructions to apply.

The selectors can be used to select elements by name as shown previously, to select elements that have the same class value (the selector is prefixed with a dot, `.`) or to select a specific element by id (the selector is prefixed with `#`).

Selectors can be combined to define stricter rules.

```css
td a.detail {
  color: green;
}
```

The preceding example applies the formatting instructions to all the `a` elements with the class value `detail` and contained in a table cell.

Style sheets can be embedded directly in a HTML document using the `style` element, or included in an external resource (a file with `.css` extension whose content only defines CSS rules). In the latter case the `link` element is used as in the following example:

```html
<link rel="stylesheet" type="text/css" href="/assets/main.css" />
```

Scripts are pieces of code embedded in the pages that can provide a richer and more interactive user experience. The programming language adopted as a standard on every modern browser is JavaScript. With JavaScript, a developer can add many functions from simple visual effects (for example interactively showing and hiding portions of the document) to more complex and useful validation functions for input controls. JavaScript relies mainly on a Document Object Model (DOM) to interact with the page contents. The DOM provides an API to create new elements in the page or to navigate and modify the existing ones.

Because haXe also targets JavaScript, it is quite obvious that in the rest of the book we will not write JavaScript code directly but always uses haXe.

The script can be embedded directly by writing the code inside a `script` element or embedding it from an external resource using the `src` attribute to point the file. In the second case, any eventual code put directly inside the script element will be discarded.

```html
<script type="text/javascript" src="/assets/main.js"></script>
```
Appendix A: Semantic HTML

The link element is used to link resources in many fashions. It can be used, as already seen, to reference one or more style sheet documents, or in conjunction with the type attribute to define the relations between a collection of documents.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Empty?</th>
<th>Frequently Used Attributes or Attribute Groups</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>style</td>
<td>false</td>
<td>i18n, type, media, title</td>
<td>Contains rules to format the current document. The style element is usually encountered inside the head element. More than one style element is allowed.</td>
</tr>
<tr>
<td>script</td>
<td>false</td>
<td>charset, src, defer, type</td>
<td>The element can contain scripting code as CData between the opening and closing tags or refer to it using an external document referred using the src attribute. In case the src attribute is used, the content between the tags will be ignored.</td>
</tr>
<tr>
<td>link</td>
<td>true</td>
<td>link, i18n, core, events</td>
<td>The link element defines a relation between the current document and an external document. On the contrary of the a attribute can be used only inside the head element of a document. One of the most common uses of the link attribute is to refer to a style sheet for the document. The element must have the attribute type with value text/css and optionally a media attribute; when absent the default value will be all.</td>
</tr>
</tbody>
</table>

Comments

It is possible to introduce comments in the HTML code. The user agents ignore the comments. To wrap a comment the syntax is:

```
<!-- comment -->
```

Where comment can be any arbitrary sequence of character.

Designers use tag to mark important sections of the document or to leave notes for future revisions in comments.
Appendix A: Semantic HTML

Attributes

So far all the base elements of the HTML standard have been described. Many of them are self-conclusive and do not need any additional information to be correctly rendered. Others require a minimum of information that is expressed in attributes. An image element (img) without a reference to the image to embed (the src attribute) is of little or no use. Many elements reuse the same attributes or a common set of attributes.

Attribute Groups

Different elements often share the same attributes that have been divided into logical groups for easy reference. Some attributes are used in more than one group at the same time and in some rare circumstances, the same attribute can have a different meaning varying with the element it is used within.

<table>
<thead>
<tr>
<th>Attribute Group</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cellalign</td>
<td>align, char, charoff, valign</td>
</tr>
<tr>
<td>cell</td>
<td>abbr, axis, colspan, headers, rowspan, scope</td>
</tr>
<tr>
<td>core</td>
<td>class, id, style, title</td>
</tr>
<tr>
<td>events</td>
<td>onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseover, onmouseout, onmouseup</td>
</tr>
<tr>
<td>form</td>
<td>accept, accept-charset, action, enctype, method, name, onreset, onsubmit</td>
</tr>
<tr>
<td>hyperlink</td>
<td>accesskey, charset, href, hreflang, name, onblur, onfocus, rel, rev, tabindex, type</td>
</tr>
<tr>
<td>i18n</td>
<td>dir, lang</td>
</tr>
<tr>
<td>image</td>
<td>alt, height, ismap, longdesc, src, usemap, width</td>
</tr>
<tr>
<td>input</td>
<td>accept, accesskey, alt, checked, disabled, height, maxlength, name, onblur, onchange, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmouseup, onmousemove, onmouseover, onmouseout, onmouseup, readonly, size, src, tabindex, type, value, width</td>
</tr>
<tr>
<td>link</td>
<td>charset, href, hreflang, rel, rev, type</td>
</tr>
<tr>
<td>object</td>
<td>archive, classid, codebase, codetype, data, declare, name, standby, tabindex, type</td>
</tr>
<tr>
<td>param</td>
<td>type, value, valuetype</td>
</tr>
<tr>
<td>table</td>
<td>border, cellspacing, cellpadding, frame, rules, summary, width</td>
</tr>
</tbody>
</table>
Appendix A: Semantic HTML

Attribute Descriptions

In the following table every attribute name is associated to a description followed by a small text describing the values that attribute can accept. This can be a reference to a value type described in the table in the “Attribute Value Types” section, or a list of acceptable values from which the author must select the most appropriate one(s).

The attributes starting with the on prefix are all related to the event model. They have a special meaning and their value is a script that is invoked when the corresponding action is executed.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbr</td>
<td>It is the abbreviation for the cell content. Value type: Text</td>
</tr>
<tr>
<td>accept</td>
<td>Specifies a list of content types that the server may accept and handle correctly when it receives them. It is useful when the form contains one or more input elements with the attribute type with value file. Value type: ContentTypeList</td>
</tr>
<tr>
<td>accept-charset</td>
<td>Specifies one or more character set used to encode the content of the form. When absent, the reserved value UNKNOWN is used. The user agent can use this value to encode the submitted content with the same character set used by the document containing the form itself. Value type: CharsetList</td>
</tr>
<tr>
<td>accesskey</td>
<td>The associated character is used as a keyboard shortcut to access the control without having to use an input device like a mouse. The keyboard button must be generally in combination with one or more control buttons to be activated; the key combination changes from browser to browser and with the operative system. Value type: Character</td>
</tr>
<tr>
<td>action</td>
<td>Specifies which URI will receive the contents of the submitted form. Value type: URI</td>
</tr>
<tr>
<td>align</td>
<td>Specifies how to align the content of a cell. Acceptable values: left: Aligns the cell content on the left-hand side. center: The cell content is centered horizontally. right: Aligns the cell content on the right-hand side. justify: Adjusts spaces within text lines so that they end evenly on the left and right margin. char: Aligns text around a specific character. Browsers usually do not implement this feature and ignore it.</td>
</tr>
<tr>
<td>alt</td>
<td>Provides an alternative text to describe an image in an &lt;img ... /&gt; element or &lt;input type=&quot;image&quot; ... /&gt; element. This attribute is very important because nonvisual browsers use it to replace the image. Value type: Text</td>
</tr>
</tbody>
</table>
## Appendix A: Semantic HTML

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>archive</td>
<td>Defines resources that are relevant to the object. Value type: A space separated list of URIs.</td>
</tr>
<tr>
<td>axis</td>
<td>A comma separated list of values. Each value represents a kind of categorization for the cell content. This attribute is rarely used because visual browsers do not give any representation of it. Value type: CData</td>
</tr>
<tr>
<td>border</td>
<td>The border width expressed in pixels. Use the value 0 to remove the border completely (implies the frame attribute value equal to void and the rules attribute value to none). A value bigger than 0 implies the frame attribute value to border and the rules attribute value to all. See also the frame and rules attributes. Value type: Pixels</td>
</tr>
<tr>
<td>cellpadding</td>
<td>The minimum vertical and horizontal space between the cell content and the cell frame. Value type: Length</td>
</tr>
<tr>
<td>cellspacing</td>
<td>The vertical and horizontal space between two adjacent cells. Value type: Length</td>
</tr>
<tr>
<td>char</td>
<td>Specifies a single character used as an axis for the alignment of several values on the same column. By default, the attribute value corresponds to the decimal separator for the current language. The char attribute works only in conjunction with the attribute align with value char. Web browsers usually simply ignore this attribute. Value type: Character</td>
</tr>
<tr>
<td>charoff</td>
<td>Used in conjunction with char, it determines the position of the axis relatively to the left margin if the text direction is left-to-right or to the right margin otherwise. Value type: Length</td>
</tr>
<tr>
<td>charset</td>
<td>Character set used to encode the linked resource. Value type: Charset</td>
</tr>
<tr>
<td>checked</td>
<td>Applies to input elements of type checkbox or radio button. It states if the control value will be submitted or not with the rest of the form. Value type: Boolean</td>
</tr>
<tr>
<td>cite</td>
<td>A URI that points to a source for the document. When used with INS and DEL, the pointed document contains the reasons why the current text has been changed. Value type: URI</td>
</tr>
<tr>
<td>class</td>
<td>A list of space-separated values. Each value is a class name. A class name is an arbitrary identifier that relates many elements to a common categorization. Value type: CData</td>
</tr>
</tbody>
</table>

*Table continued on following page*
### Appendix A: Semantic HTML

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>classid</code></td>
<td>Is used to specify the location of an object implementation. It can be used in conjunction with the data attribute or alone depending on the type of the object. Value type: URI</td>
</tr>
<tr>
<td><code>codebase</code></td>
<td>Defines the base URI to use with the attributes <code>classid</code>, <code>data</code>, and <code>archive</code> when their value is a relative path. When omitted, its value defaults to the base URI relative to the current document. Value type: URI</td>
</tr>
<tr>
<td><code>codetype</code></td>
<td>Defines the expected content type when downloading the object specified by <code>classid</code>. Value type: <code>ContentType</code></td>
</tr>
<tr>
<td><code>cols</code></td>
<td>Defines the width of a control measured in average characters width. Lines exceeding the maximum length are wrapped or some scrolling mechanism is activated. Value type: Number</td>
</tr>
<tr>
<td><code>colspan</code></td>
<td>The number of columns that is spanned by the current cell. The default value is 1; the value 0 means that the cell spans all columns from the current to the last in the table section (<code>thead</code>, <code>tbody</code>, <code>tfoot</code>, or <code>table</code> if no section is defined). Value type: Number</td>
</tr>
<tr>
<td><code>content</code></td>
<td>Specifies the value of a property. It is used in pair with the <code>name</code> or the <code>http-equiv</code> attributes inside a <code>meta</code> element. Value type: <code>CData</code></td>
</tr>
<tr>
<td><code>coords</code></td>
<td>Each numeric value in the list represents a coordinate. They assume a different meaning conforming to the <code>shape</code> attribute. Value type: A comma separated list of <code>Number</code> values. The coordinates will represent respectively: rect: left, top, right, bottom. circle: center-x, center-y, radius. poly: x1, y1, x2, y2 ..., xN, yN. The polygon is always a closed shape.</td>
</tr>
<tr>
<td><code>data</code></td>
<td>Is the address for the resource to load in an <code>object</code> element. Value type: URI</td>
</tr>
<tr>
<td><code>datetime</code></td>
<td>The date and time when a modification has been made to the document. Value type: <code>DateTime</code></td>
</tr>
<tr>
<td><code>declare</code></td>
<td>When present, it makes the current <code>object</code> element a declaration only. The object must be instantiated by a subsequent object definition referring to this element. Value type: <code>Boolean</code></td>
</tr>
<tr>
<td><code>defer</code></td>
<td>When present the user agent receives a hint that the script will not generate any document content and, thus, it can continue with the parsing and rendering action. Value type: <code>Boolean</code></td>
</tr>
</tbody>
</table>
### Appendix A: Semantic HTML

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| dir            | Used to set the left-to-right or right-to-left direction for directionally neutral text like Hebrew.  
Acceptable values:  
LTR: left-to-right direction  
RTL: right-to-left direction |
| disabled       | The corresponding control element is rendered but the user cannot alter its value. The only way to change the disabled attribute is using a script. The value associated to a disabled control will not be sent to the server on form submission.  
Value type: Boolean |
| enctype        | Specifies the content type used to submit the form contents when the method attribute is set to post. When absent, the default value is application/x-www-form-urlencoded; it should be changed to multipart/form-data when an input element with attribute type="file" exists in the form.  
Value type: ContentType |
| for            | Explicitly associates a label element to a control using the control id as value.  
Value type: IDRef |
| frame          | Determines which sides of the frame containing the table will be visible. The default value is void. The frame and the rules attributes affect the borders between rows and columns. See also the rules and border attributes.  
Acceptable values:  
void: No border is visible.  
above: Only the top side is visible.  
below: Only the bottom side is visible.  
hsides: Both the top and the bottom sides are visible.  
vsides: Both the right and the left sides are visible.  
lhs: Only the left-hand side is visible.  
rhs: Only the right-hand side is visible.  
box: Visible on all the four sides.  
border: Visible on all the four sides. |
| headers        | The attribute value is a list of header cells that provide heading information for the current cell. The list is composed of id values (see the id attribute).  
Value type: IDREFS |
| height         | Designates the height of the associated element.  
Value type: Length |
| href           | URI for the linked resource.  
Value type: URI |
| hreflang       | Determines the language of a resource at the end of a link.  
Value type: LanguageCode |
| http-equiv      | It is alternative to the name attribute. HTTP servers use this attribute to gather information for HTTP response message headers.  
Value type: Name |

*Table continued on following page*
# Appendix A: Semantic HTML

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| id             | Is a unique identifier for the element within the whole document. It is mainly used to identify the element for styling and scripting.  
Value type: ID |
| ismap          | When present, it indicates that the current image element will be treated on the server side as a map. This user agent will send to the server two variables with the pointer device coordinates. The coordinates are in pixel and relative to the top-left corner of the image element that the map is applied to. Their values will be stored in two variables with the same name as the name attribute with the suffix `.x` and `.y`.  
Value type: Boolean |
| label          | Describes the group of options.  
(optgroup)  
Value type: Text |
| label (option) | Provides a short description of the option. The attribute may be used by some agents as a replacement for the content of the option element.  
Value type: Text |
| lang           | Specifies a language for the element content and attributes. This attribute is usually defined when the language used in the element is not the same of the container element.  
Value type: LanguageCode |
| longdesc       | Specifies a link to a document that contains a long description of the image resource.  
Value type: URI |
| maxlength      | Used in conjunction with a control of type text or password, it sets the maximum number of allowed characters.  
Value type: Numeric |
| media          | Specifies the intended medium for the style information. It is possible to specify a list of comma separated values.  
Acceptable values:  
all: Suitable for any device type.  
screen: Applies to graphic enabled devices (standard modern browsers).  
print: Intended for printed documents.  
handheld: Intended for palmtop devices (low resolution, small display, limited colors and bandwidth).  
tv: Addresses television-type devices (low resolution and limited user interface interaction).  
aural: Intended for audio devices (speech synthesizers).  
tty: Intended for devices that uses a fixed-pitch character grid such as terminals with no or limited graphic support.  
projection: Intended for image/video projectors.  
braille: Intended for Braille devices. |
Appendix A: Semantic HTML

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| **method**     | Specifies which HTTP method will be used to submit the form contents. Acceptable values:  
|                | *get*: Form contents are encoded and appended to the URI.  
|                | *post*: Form contents are sent in the header of the request.  |
| **multiple**   | Specifies if the select element allows multiple selections. When multiple values are selected, the server will receive more than one variable with the same name and different values.  
|                | Value type: Boolean |
| **name** (a)   | The element acts as an anchor. A hyperlink can reference an anchor appending to the URI the cardinal symbol # followed by the anchor name. When a link to an anchor is followed, the browser will automatically scroll the page to show the anchor on the top of the page if possible.  
|                | Value type: Cdata |
| **name** (input, select, textarea, form) | It is a name so that the element can be referenced for styling and scripting, it exists only for backward compatibility reason and should be replaced by id. In the case of form controls, its value will be the variable name sent to the server on form submission. The id attribute can't be used for the same purpose.  
|                | Value type: CData |
| **nohref**     | When present, it indicates that the region has no associated link.  
|                | Value type: Boolean |
| **onblur**     | The onblur event occurs when the element is active and loses the focus.  
|                | Value type: CData |
| **onchange**   | The onchange event occurs when the control loses the focus and its value has been modified since gaining focus.  
|                | Value type: CData |
| **onclick**    | The associated script is invoked when the pointing device is clicked once over the element.  
|                | Value type: CData |
| **ondblclick** | The associated script is invoked when the pointing device is clicked twice over the element in a short lapse of time.  
|                | Value type: CData |
| **onfocus**    | When the element obtains the focus (for example: an input text box has the current), the corresponding script is executed.  
|                | Value type: CData |
| **onkeydown**  | The associated script is invoked when a keyboard button is pushed.  
|                | Value type: CData |
| **onkeypress** | The onkeypress event occurs when a keyboard button is pressed.  
|                | Value type: CData |
| **onkeyup**    | The associated script is invoked when a keyboard button is released.  
|                | Value type: CData |

Table continued on following page
## Appendix A: Semantic HTML

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>onload</td>
<td>The <code>onload</code> event fires when the user agent finishes loading the whole document and its dependencies. This event is associated with the body element. Value type: CData</td>
</tr>
<tr>
<td>onmousedown</td>
<td>The <code>onmousedown</code> event fires when the pointing device is pressed over the element. Value type: CData</td>
</tr>
<tr>
<td>onmousemove</td>
<td>The <code>onmousemove</code> event occurs when the pointing device moves over the element. Value type: CData</td>
</tr>
<tr>
<td>onmouseout</td>
<td>The <code>onmouseout</code> event fires when the pointing device moves away from the element boundaries. Value type: CData</td>
</tr>
<tr>
<td>onmouseover</td>
<td>The <code>onmouseover</code> event fires when the pointing device moves onto the element. Value type: CData</td>
</tr>
<tr>
<td>onmouseup</td>
<td>The <code>onmouseup</code> event fires when the pointing device is released over the element. Value type: CData</td>
</tr>
<tr>
<td>onreset</td>
<td>The <code>onreset</code> event fires when a reset action has occurred. Value type: CData</td>
</tr>
<tr>
<td>onselect</td>
<td>The <code>onselect</code> event occurs when a user selects some text in a text field. Value type: CData</td>
</tr>
<tr>
<td>onsubmit</td>
<td>The <code>onsubmit</code> event occurs when a form is submitted. Value type: CData</td>
</tr>
<tr>
<td>onunload</td>
<td>The <code>onunload</code> event occurs when a document is removed by its enclosing window. This event is associated with the body element. Value type: CData</td>
</tr>
<tr>
<td>profile</td>
<td>Points to a resource that describes how a client must interpret the header meta information contained in the <code>meta</code> elements inside the <code>head</code> element. Value type: URI</td>
</tr>
<tr>
<td>readonly</td>
<td>The corresponding control element is visualized but the user cannot alter its value. Values associated to read-only controls are sent to the server on form submission. The only way to change the disabled attribute is using a script. Value type: Boolean</td>
</tr>
<tr>
<td>rel</td>
<td>Describes the relationship between the current document and the one pointed by the <code>href</code> attribute. Value type: LinkTypes</td>
</tr>
<tr>
<td>rev</td>
<td>It acts as the rel attribute but reverses the relationship direction. Value type: LinkTypes</td>
</tr>
</tbody>
</table>
### Appendix A: Semantic HTML

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| **rows**      | Defines the number of visible text lines. In case the user inputs more lines than permitted, the browser will allow the scrolling of the area.  
Value type: Number |
| **rowspan**   | The number of rows that is spanned by the current cell. The default value is 1; the value 0 means that the cell spans all rows form the current to the last in the table section (thead, tbody, tfoot, or table if no section is defined).  
Value type: Number |
| **rules**     | Specifies which rules will appear between rows and columns within a table. The default value is null. See also the frame and border attributes.  
Acceptable values:  
none: No rules are displayed.  
groups: Rules appear only between row groups (thead, tfoot and tbody) and column groups (colgroup and col).  
rows: Rules appear only between rows.  
cols: Rules appear only between columns.  
all: Rules appear between all rows and columns. |
| **scheme**    | This attribute names a scheme to be used to interpret the property’s value. It is used in association with the profile attribute.  
Value type: CData |
| **scope**     | Specifies the set of cells for which the current cell header provides heading information. It is used alternatively to the headers attribute in simple tables.  
Acceptable values:  
row: Provides header information for the rest of the row.  
col: Provides header information for the rest of the column.  
rowgroup: Provides header information for the rest of the row group.  
colgroup: Provides header information for the rest of the col group. |
| **selected**  | When present, the current option is selected.  
Value type: Boolean |
| **shape**     | Defines the type of region that the current area element will represent.  
Acceptable values:  
default: Specifies the entire region.  
rect: Defines a rectangular region.  
circle: Defines a circular region.  
poly: Defines a polygonal region. |
| **size**      | Specifies the initial width of a control expressed in pixels for all controls but text and password; in that case it is expressed in number of characters. Using style rules to fix the size of a control is usually a better option.  
Value type: Number |
| **span**      | The number of columns affected by the attributes of the col or colgroup elements. Default value is 1.  
Value type: Number |

*Table continued on following page*
## Appendix A: Semantic HTML

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>Points to an external resource to use in the document. A user agent with graphic capabilities will render the pointed image in place of elements like <code>&lt;img ... /&gt;</code> or <code>&lt;input type=&quot;img&quot; ... /&gt;</code>. Value type: URI</td>
</tr>
<tr>
<td>standby</td>
<td>Defines a text message that the user agent may display while loading the required resources for the current object. Value type: Text</td>
</tr>
<tr>
<td>style</td>
<td>Contains information about the format of the element. It is used to alter the presentation of the element. Value type: CDATA</td>
</tr>
<tr>
<td>summary</td>
<td>Provides a summary for the table purpose and structure. It is mainly used by nonvisual browsers. Value type: Text</td>
</tr>
<tr>
<td>tabindex</td>
<td>Controls and active elements such as hyperlinks may be navigated using the keyboard Tab button. Each time it is pressed the next control in sequence will obtain the focus. The <code>tabindex</code> attribute enable you to specify the desired sequence between controls. When two controls have the same <code>tabindex</code> value, the first appearing in the document code will have the priority. Value type: Numeric</td>
</tr>
<tr>
<td>title</td>
<td>Offers additional information about the element. The browsers usually render it as a tooltip. Value type: Text</td>
</tr>
<tr>
<td>type (input)</td>
<td>Determines the kind of control to render on the page. Acceptable values: text: A single line text-input control. password: Same as text but hides the typed characters replacing them with an asterisk (or another masking symbol). checkbox: Creates a checkbox. The value of the attribute value along with the control name will be sent only if the control is checked. radio: Creates a radio button. Only the value of checked radio controls will be sent to the server on form submission. submit: Creates a submit button. It finalizes the form submission. image: Creates a graphical submit button. The submitted data will also contain the coordinates of the pointing device at the moment of the click action. See the definition for the <code>ismap</code> attribute for further information. reset: Creates a reset button. A reset button restores the default values for each control in the current form. button: Creates a standard push button. Actions must be associated to the button using a script and leveraging on the click event. hidden: Creates a hidden control. This control has no visual representation but its value will be submitted to the server along with the other controls. file: Creates a file select control. It is used to upload a file to the server.</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>type (a, object, param, link, script, style)</td>
<td>Defines the content type of the linked resource.</td>
</tr>
<tr>
<td></td>
<td>Value type: ContentType</td>
</tr>
<tr>
<td>usemap</td>
<td>Associates a map with an image element. A <code>map</code> element defines the image map characteristics. The <code>usemap</code> attribute value must match the name attribute of the <code>map</code> element.</td>
</tr>
<tr>
<td></td>
<td>Value type: URI</td>
</tr>
<tr>
<td>valign</td>
<td>Defines the vertical alignment of the cell content.</td>
</tr>
<tr>
<td></td>
<td>Acceptable values:</td>
</tr>
<tr>
<td></td>
<td>top: The cell content is located on the top of the cell.</td>
</tr>
<tr>
<td></td>
<td>middle: The cell content is centered vertically.</td>
</tr>
<tr>
<td></td>
<td>bottom: The cell content is positioned at the bottom of the cell.</td>
</tr>
<tr>
<td></td>
<td>baseline: The content of all the cells in the same row with this attribute and value, have the first line of text on the same axis.</td>
</tr>
<tr>
<td>value (input)</td>
<td>Contains the value that will be sent on form submission. A control that is created with a value attribute will show its content as a default value to the user.</td>
</tr>
<tr>
<td></td>
<td>The attribute is optional except when the attribute type is radio or checkbox.</td>
</tr>
<tr>
<td></td>
<td>Value type: CData</td>
</tr>
<tr>
<td>value (param)</td>
<td>Defines the value for the current <code>param</code> element.</td>
</tr>
<tr>
<td></td>
<td>Value type: CData</td>
</tr>
<tr>
<td>valuetype</td>
<td>Specifies the type of the <code>value</code> attribute. The default value is data.</td>
</tr>
<tr>
<td></td>
<td>Acceptable values:</td>
</tr>
<tr>
<td></td>
<td>data: The value will be passed to the object as a string.</td>
</tr>
<tr>
<td></td>
<td>ref: The value defines a URI where the real value may be located.</td>
</tr>
<tr>
<td></td>
<td>object: The value is an object identifier that matches the name of another object element in the current document.</td>
</tr>
<tr>
<td>width (col, colgroup)</td>
<td>The value refers to a default width for each column in the group.</td>
</tr>
<tr>
<td></td>
<td>Value type: One or more space-separated Length values. Accepts also the special value 0* which means that the column will have a width as large as the column contents.</td>
</tr>
<tr>
<td>width (image, table)</td>
<td>It determines the width of the associated element.</td>
</tr>
<tr>
<td></td>
<td>Value type: Length</td>
</tr>
</tbody>
</table>
## Appendix A: Semantic HTML

### Attribute Value Types

The attribute value can be composed of fixed keywords or arbitrary text; the following table defines the acceptable value types used in many common attributes.

<table>
<thead>
<tr>
<th>Attribute Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Some attributes may exist without any value at all; the element gains information by the presence or absence of the attribute itself. In HTML, an attribute without a value is legal but in XHTML it is not (XML has a stricter syntax over HTML). To be XHTML compatible the attribute value will be the same as the attribute name (for example: <code>selected=&quot;selected&quot;</code>). To express the false condition of a Boolean attribute just remove the attribute and its value.</td>
</tr>
<tr>
<td>CData</td>
<td>It is a sequence of arbitrary characters. Some characters can create ambiguous situations (such as using double-quotes in a double-quoted attribute value) and must be replaced by their entity equivalent. The more frequent and problem prone characters/entities are: &amp;lt; representing the <code>&lt;</code> sign. &amp;gt; representing the <code>&gt;</code> sign. &amp;quot; representing the &quot; mark.</td>
</tr>
<tr>
<td>Character</td>
<td>A single character value.</td>
</tr>
<tr>
<td>Charset</td>
<td>A character set as defined by the ISO10646 and Unicode standards.</td>
</tr>
<tr>
<td>CharSetList</td>
<td>A space- and/or comma separated list of character set values.</td>
</tr>
<tr>
<td>ContentType</td>
<td>The attribute value gives indications on the format of the related document. The acceptable values are defined in the MIME type codification (<a href="http://www.w3.org/TR/html401/references.html#ref-MIMETYPES">http://www.w3.org/TR/html401/references.html#ref-MIMETYPES</a>) and include values like <code>text/html</code>, &quot;image/png&quot; and &quot;text/css&quot;.</td>
</tr>
<tr>
<td>ContentTypeList</td>
<td>A comma separated list of ContentType values.</td>
</tr>
<tr>
<td>DateTime</td>
<td>Defines a date and time value according to the ISO8601 standard (<a href="http://www.w3.org/TR/html401/references.html#ref-ISO8601">http://www.w3.org/TR/html401/references.html#ref-ISO8601</a>). The date format is expressed like this: <code>YYYY-MM-DDThh:mm:ssTZD</code> where: <code>YYYY</code>: four-digit year <code>MM</code>: Two-digit month (01=January ...) <code>DD</code>: Two-digit day of month (01 through 31) <code>T</code>: Is a fixed literal used to separate the date component from the time component. <code>hh</code>: Two digits for hour (00 through 23) <code>mm</code>: Two digits for minute (00 through 59) <code>ss</code>: Two digits for second (00 through 59) <code>TZD</code>: Time zone designator. It can assume the value <code>Z</code> for UTC (Coordinated Universal Time), <code>+hh:mm</code> or <code>-hh:mm</code> for hours and minutes ahead or behind the UTC.</td>
</tr>
</tbody>
</table>
### Appendix A: Semantic HTML

<table>
<thead>
<tr>
<th>Attribute Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID and Name</td>
<td>An alphabet letter (both lower- or upper-case) followed by any number of letters, digits, hyphens (-), underscores (_), colons (:) and periods (.)</td>
</tr>
<tr>
<td>IDRef</td>
<td>A reference to an ID that exists associated to an element in the current document</td>
</tr>
<tr>
<td>IDRefs</td>
<td>A space separated list of IDRef values</td>
</tr>
<tr>
<td>LanguageCode</td>
<td>The language is expressed using the standard RFC1766 (<a href="http://www.ietf.org/rfc/rfc1766.txt">http://www.ietf.org/rfc/rfc1766.txt</a>) Acceptable values are like en (English) and en-US (U.S. version of English)</td>
</tr>
<tr>
<td>Length</td>
<td>A measure expressed in pixels or as a percentage of the available horizontal or vertical space. When the measure is in pixels no suffix is required otherwise the percent symbol (%) must be used</td>
</tr>
<tr>
<td>LinkType</td>
<td>Adds semantic value to the pointed resource. The attribute can assume one of the following values: Alternate: A substitutive version of the current document. Used with the lang attribute, it implies a translation of the document. Used with the media attribute, it implies a version of the document for a different support. Stylesheet: Calls an external document that contains formatting rules. Start: Refers to the first document in a collection. Next: Refers to the next document in a collection as defined by the author. Prev: Refers to the previous document in a collection as defined by the author. Contents: Refers to a document that serves as a table of contents (TOC). Index: Refers to a document providing an index for the current document. Glossary: Refers to a document that provides a glossary of terms that pertain to the current document. Copyright: Refers to a copyright statement for the current document. Chapter: Refers to a document that acts as a chapter in a collection. Section: Refers to a document that acts as a section in a collection. Subsection: Refers to a document that acts as a subsection in a collection. Appendix: Refers to a document that acts as an appendix in a collection. Help: The referred document contains help support (more information, links to supporting resources ...). Bookmark: Refers to a key point in a long document</td>
</tr>
<tr>
<td>LinkTypes</td>
<td>A list of space-separated LinkType</td>
</tr>
<tr>
<td>Number</td>
<td>The value is composed by one or more digits representing a positive integer value</td>
</tr>
<tr>
<td>Pixels</td>
<td>Is a positive integer value indicating a measure expressed in pixels</td>
</tr>
<tr>
<td>Text</td>
<td>Is meant to be a human readable attribute value. The restrictions for CData apply the same here</td>
</tr>
<tr>
<td>URI</td>
<td>A web address as formalized in the RFC2396 (<a href="http://www.ietf.org/rfc/rfc2396.txt">http://www.ietf.org/rfc/rfc2396.txt</a>)</td>
</tr>
<tr>
<td>Width</td>
<td>The measure can be expressed in pixels or in percent; the latter case must be suffixed with the percent symbol (%)</td>
</tr>
</tbody>
</table>
A
abstract classes, 119–120
abstract methods, 119–120
abstract value types, 35–36
dynamic type, 35–36
garbage collection, 572–573
Neko and C/C++, 563–565
null type, 35
void type, 35
accessor methods, with dynamic keyword, 113
Action Message Format (AMF) server, 426
Flash-to-AMF server connection, 434
ActionScript
and Flash, 316
compared to haXe. See ActionScript/haXe comparison
Screen Weaver HX extensions, 523–524
synchronous communication in, 523–524
version 3 and Flex 2, 344
ActionScript/haXe comparison
classes, 357–359
constant values/special values, 346–347
data types, 344–346
functions, top-level, 347–352
keywords, 354–358
operators, 352–353
properties, 356
statements, 354–358
structures, 354–358
trace, 181
ActionScript Virtual Machine 2, 316
ActionStep library, 369
add
lists, 47
strings, 71–72
addChar, 71–72
addition (+), binary operator, 55
addSub, 71
Adobe Flash CS3, benefits to use, 363
Adobe Flex library, 369
AIR, 510
AJAX, 228, 414–422
auto-complete control, building, 417–422
functions of, 414
HTML panel content, updating, 416–417
alloc_root function, 571–572
alloc function, 572
alternation, regular expressions, 220
anchors, regular expressions, 217–218
AND (&), bitwise operator, 56, 62
AND (&&), logical comparison operator, 59
animation, Neko Media Engine (NME), 549
anonymous keyword, 28
anonymous objects, 126–127
declaring, syntax for, 126
typedef, use with, 128–130
Apache web server
installing on Linux Ubuntu, 233–234
installing on Windows, 232–233
application programming interface (API)
Flash API classes, 359–361
Flash drawing API, 337–342
haXe API, 206–208
reflection API, 464–469
Remoting API communication methods,
427–428
serialization API, 473–474
arctic project, 146
arithmetic operators. See binary operators
array(s), 40–45
adding/removing items from, 42–43
array access operator, 352
array fields/descriptions, 41–42
concatenating, 45
converted to strings, 68
converting string to, 67
copying, 44–45
data type, changing, 41
functions of, 41
hash tables, 48–50
initializing with literal values, 41
instantiating, 41
iterator, creating, 98
lambda class for modifying, 96–98
lists, 46–48
multidimensional, 45–46
Neko and C/C++, 567–568
sort method, 96
typing rules for, 40
values, conversion methods, 43–44
array method, lambda class, 98
Array string, 66
aspell project

assignment operators
types of, 56
uses of, 59–60
asterisk, block commenting, 37
AsWing library, 369
asynchronous communications, 426
AsyncAdapter, 437
AsyncConnection, 432–434, 440
AsyncDebugConnection, 437
AsyncProxy, 438
Flash-Neko communication, 518–520
attr expressions, Templo, 262
attributes, HTML, 589–601
attribute descriptions, 590–599
attribute groups, 589
attribute value types, 600–601
attributes, XML
accessing, 457–458
checking, 461–462
Author class, SPOD system, 287–288, 296–297, 299
AuthorManager class, SPOD system, 300–301
auto-complete control, building, AJAX technique, 417–422

back-references, regular expressions, 214, 220
backward slash, escaped characters, 33–34
baseEncode, 69
bfailure function, 574
BinaryNode class, 123
binary operators, 56–58
increment/decrement, 57–58
modulo operator, 57
order of precedence, 56–57
Bit Block Transfer, 540
bitwise mathematics, utility of, 60
bitwise operators, 60–63
filtering bits, 62–63
shifting bits, 61–62
block commenting, 37
blog entry, constructors, use of, 108–110
bool
standard data type, 28
type data conversion, 39
Boolean(s)
functions of, 33
restrictions in haXe, 33
boot classes, magic, 475
break, with loops, 84–85
bseDecode, 69
buffers, display buffer, flipping, 535–536
builtins, Neko, 477
buttons
Flash user interface, 376, 377
nGui, 488–489
byte(s)
bytecode, Neko, 156–157
extracting from stream, 306
readBytes method, 306
writeBytes method, 308–309
CalcProxy class, 438
calculation expressions, 253
callback
macros in templates, 256–257
nGui list control, 491–492
callStack function, 191–193, 196–199
carriage return, escaped characters, 34
case keyword, and switch statements, 34
casting, 38–40
cast function, operation of, 38–39
conversion methods, types of, 39–40
type of value, comparing, 40
unsafe cast, 38–39
catchall block, for exceptions, 189
caurinaTween project, 146
C/C++, with Neko. See Neko and C/C++
cell, float to int conversion, 66
Chapter class, SPOD system, 296–297, 299–300
ChapterManager class, SPOD system, 300–301
class(es), 102–103, 122–126
abstract classes, 119–120
ActionScript/haXe comparison, 357–359
advanced, 122
exception class, 194–199
extensions, use of, 133–134
extern classes, 150, 160
functions, 102–103
functions of, 102
child controls, Flash user interface, 376–381
chr, type data conversion, 39
check class, XML, 458
carriage return, escaped characters, 34
character classes
non-alphanumeric characters, 217
range of values, 216
regular expressions, 215–217
syntax for, 216
cast function, operation of, 38–39
conversion methods, types of, 39–40
type of value, comparing, 40
unsafe cast, 38–39
catchall block, for exceptions, 189
caurinaTween project, 146
C/C++, with Neko. See Neko and C/C++
cell, float to int conversion, 66
Chapter class, SPOD system, 296–297, 299–300
ChapterManager class, SPOD system, 300–301
character(s)
patterns, regular expressions, 214–215
readChar method, 305–306
serialization, 473–474
writeChar method, 308
character classes
non-alphanumeric characters, 217
range of values, 216
regular expressions, 215–217
syntax for, 216
character sets, regular expressions, 217
charAt, 66–67
charCodeAt, 66–67
Check class, XML, 458
child controls, Flash user interface, 376–381
chr, type data conversion, 39
class(es), 102–103, 122–126
abstract classes, 119–120
ActionScript/haXe comparison, 357–359
advanced, 122
exception class, 194–199
extensions, use of, 133–134
extern classes, 150, 160
functions, 102–103
functions of, 102
implementing, 123
name, elements of, 102
public and private, 102
syntax for, 102
template class, 252
type parameters, 123–126
variables, 102–103
class blocks, HaXe program structure, 23
class functions, 85–90
functions of, 85
structure of, 85–86
class keyword, 28
clear, lists, 47
clear function, trace, 184
code
documenting, 166–170
libraries, 145–162
packages, 139–145
resources, 163–166
unit testing, 170–177
collection(s)
functions in, 94–96
items, accessing directly, 83
looping over, 81–82
collisions, detecting, Neko Media Engine (NME), 543–545
color key, Neko Media Engine, 540
commenting
block commenting, 37
HTML, 588
line commenting, 37
regular expressions, 220–221
comparison expressions, templates, 253
comparison operators, logical comparison operator, 58–59
compiler directives, functions of, 6
concatenation, arrays, 41, 45
conditional(s), regular expressions, 220–221
conditional statements, 74–79
if statement, 74–77
switch statement, 77–78
values, returning from, 78–79
cfg project, 148
cfgStr, 68
confirm message box, SysTools, 530
Connection class, Neko database support, 277–278
constant values, ActionScript/haXe comparison, 346–347
constant variables, 31
constraint parameters, 125–126
constructor(s), 108–110
constructor arguments, 136–138
enum arguments, 136–138
of instance variables, 108–110
overriding in inheritance, 118
syntax for, 108
containers
Flash user interface, 376–377
nGui, 487–488
continue, with loops, 84–85
Control class, nGui, 485–487
controls
auto-complete control, AJAX technique, 414–422
Flash user interface, 375–376
HTML, 585–586
nGui library, 482
copying, arrays, 41, 44–45
createDirectory method, Neko, 303
createdOn, 112, 122
cryptographic hashes, 224
databases, Neko. See Neko database support
data conversion, casting, 38–40
data types, ActionScript/haXe comparison, 344–346
date/time, 50–54
components of, retrieving, 52–53
date method/descriptions, 50–51
date objects, creating, 51–52
delta( ) function, 53
format( ) function, 53–54
getMonthDays(), 54
tokens/descriptions, 54
dBName method, 281
dcxml project, 146
debugging, 179–199
exception class, 194–198
exceptions, 185–194
trace, 179–185
declarations, variables, 29–30
decrement (–), binary operator, 55, 57–58
default
haXe standard library, 145
variable value modifier, 111
default case, 135
default keyword, and switch statements, 78
delayed actions, timer, 223
DelayedConnection, 437
deleteDirectory method, Neko, 303
deleteFile method, Neko, 303
delimiters, string(s), 67
delta function, dates, 53
dialogs, SysTools, 529–530
directory management
create/delete directory, 303
Neko, 303–304
reading from directory, 303–304
display buffer, flipping, 535–536
division (/)
binary operator, 55
divide by zero issue, 64–65
modulo operator, 55, 57
<table>
<thead>
<tr>
<th><strong>DNS (Domain Name Server)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>DNS (Domain Name Server), 227</td>
</tr>
<tr>
<td>documentation of code, 166–170</td>
</tr>
<tr>
<td>offline, 167–169</td>
</tr>
<tr>
<td>online, 169–170</td>
</tr>
<tr>
<td>document definition elements, HTML, 577–578</td>
</tr>
<tr>
<td>definitions in file, 393</td>
</tr>
<tr>
<td>typedef declarations, 394–396</td>
</tr>
<tr>
<td>Document Object Model (DOM), 392–396</td>
</tr>
<tr>
<td>definitions in file, 393</td>
</tr>
<tr>
<td>typedef declarations, 394–396</td>
</tr>
<tr>
<td>Document Type Definition (DTD), 229–230</td>
</tr>
<tr>
<td>Dojo toolkit, 399</td>
</tr>
<tr>
<td>dot expressions, 253–254</td>
</tr>
<tr>
<td>double quotes, escaped characters, 34</td>
</tr>
<tr>
<td>dowhile loop, 80–81</td>
</tr>
<tr>
<td>drag events, 515</td>
</tr>
<tr>
<td>drawing</td>
</tr>
<tr>
<td>to display, Neko Media Engine (NME), 538–540</td>
</tr>
<tr>
<td>Flash API, 337–342</td>
</tr>
<tr>
<td>dynamic</td>
</tr>
<tr>
<td>functions, 87–88</td>
</tr>
<tr>
<td>keyword, accessor methods with, 113</td>
</tr>
<tr>
<td>links, Flash, 154–156</td>
</tr>
<tr>
<td>standard data type, 28</td>
</tr>
<tr>
<td>variable value modifier, 111</td>
</tr>
<tr>
<td>dynamic type, 127–133</td>
</tr>
<tr>
<td>parameters, 127–128</td>
</tr>
<tr>
<td>typedef, 128–131</td>
</tr>
<tr>
<td>using, 35–36</td>
</tr>
<tr>
<td>dynamic typing, variables, 29</td>
</tr>
</tbody>
</table>

---

**E**

- **element**
  - Flash user interface, 374–375
  - HTML, 226–227
- **else keyword, and if statement, 75–76**
- **encoding, strings for web, 71**
- **endsWith**, 69–70
- **enum, 134–136**
  - Condition, 136
  - constructor arguments, 136–138
  - keyword, 28
  - and packages, 143
  - and switch, 135, 143
  - syntax for, 134
- **equal (==), 56, 58–59**
- **EReg class, 210–214**
- **error handling, Neko and C/C++, 573–574**
- **escaped characters, string content, 33–34**
- **events**
  - Flash, 328–331
  - Flash user interface, 371–373
  - hxGtk library, 502–503
  - Neko Media Engine (NME), 549–553
  - Screen Weaver HX window events, 515–516
- **exception(s), 185–194**
  - catchall block, 189
  - catching exceptions, 187–189
  - complex, dealing with, 189–190
  - elements of, 185–186
  - exception class, 194–199
  - exceptionStack function, 193–194, 196
  - manifestation of, 186–187
  - throwing exceptions, 188
  - unexpected, catching, 191
- **exception class, 194–199**
  - code for, 197–199
  - creating, 194–195
  - debug functionality, 195–197
  - generic output function, 197
- **excerpt variable, 112**
- **execute method, Web development, 242**
- **exists method, 303**
  - hash method, 49
- **explicit import, 142–143**
- **expressions, regular. See regular expressions**
- **extensions, 133–134**
  - with classes, 133–134
  - with typedef, 134
- **external libraries, 150–162**
  - extern classes, 150, 160
  - Flash, 151–156
  - JavaScript, 159–162
  - Neko, 156–158
- **Ext library, 400**

---

**F**

- **failure function, 573–574**
- **fash package project, 146**
- **Fast class, XML, 455–456**
- **fhx project, 146**
- **field access operator, 353**
- **FieldLayout, Flash user interface, 381**
- **file(s), database**
  - appending to existing file, 308
  - deleting, 303
  - directory management, 303–304
  - File class, Neko, 304–311
  - FileSystem class, Neko, 301–302
  - path strings, 302–303
  - read data directly, 307
  - read data from stream, 305–306
  - remote files, accessing, 311
  - traversing stream, 309–310
  - write data to stream, 307–309
- **FileInput stream, 304–305**
- **FileOutput stream, 304–305**
- **fill expression, Templo, 265**
- **fill List function, 440**
- **filters**
  - filtering bits, 62–63
lists, filter method, 95
XML, rules, 462–463
Firebug
Firebug, haXe trace redirected to, 389–390
FireFox
Firebug, 389–390
Web Developer extension, 389
first, lists, 47
flags, Screen Weaver HX window flags, 513
Flash, 315–385
API classes, 359–361
compiling to, 5–6
development of, 316
drawing API, 337–342
dynamic links, 154–156
events, 328–331
extending methods, for MovieClip class, 342–344
external library, 151–156
FlashJsConnection, 435
Flash-Neko communication, 517–524
Flash-to-Action Message Format (AMF) Server connection, 434
Flash-to-JavaScript connection, 429–431
functions of, 315
to haXe. See Action Script/haXe comparison.
.hxml file compiled to, 18
JavaScript-to-Flash connection, 432
LocalConnection, 435
magic, 477–478
Remoting API communication methods, 427–428
and Screen Weaver HX, 517–524
slideshow, 331–337
SocketConnection, 436
static links, 152–154
SWFMill, 364–369
SWF output, 6
switches, 24
TicTacToe Game, 443–452
trace output, 180
Unicode support, 221–222
user interface, creating. See Flash user interface version 9 differences, 156
Flash movies
clip, creating in version 9, 321–322, 324–328
clip, creating in versions 6 through 8, 319–321, 324–330
duplicating, 321
embedding in page, sample code, 318
embed technique, problems/solutions, 318–319
file extension (.swf), 316
images in, 323–324
mask, use on movie clip, 342–344
MovieClip class, 319–322, 342–344
sounds, 325
stage and timeline, 316–317
text in, 322–323
values, loading, 325
variables, loading, 326–327
XML, loading, 327–328
flash package, 145
Flash Player
Detection Kit, 319
express install, 319
parameters, determination of, 318
purpose of, 315, 317
Flash user interface, 369–385
containers, 376–377
controls, 375–376
element fields, 374–375
events, 371–373
file structure, 370–371
Flash 9 implementation, 381
layout constructors, 374
library, building, 369–371
tree editor, example of, 381–384
Flex 2, 344
Flex Builder IDE, compiling to Flash, 6
flex project, 146
float
elements of, 32
to int conversion, 65–66
standard data type, 28
floating-point numbers, double precision, 32
floor, float to int conversion, 65–66
fmod project, 146
fold, lambda class, 98
fonts, True Type, embedding in SWFMill movie clips, 368–369
fonttools project, 146
format( ) function, dates, 53–54
forms
HTML, 585–586
HTML, data validation, 404–414
forward slashes, block commenting, 37
framework_delfiweb project, 147
free_root function, 572
fromTime, 51
front controller, Neko as, 238–239
FTP (file transfer protocol), 226
fullpath method, 302
function(s), 85–98
ActionScript/haXe comparison, 347–352
of classes, 102–103
class functions, 85–90
in collection objects, 94–96
dynamic functions, 87–88
elements of, 85
function type inference, 87
identifiers sequence, 108
instance functions, 104–107
JavaScript, 160
lambda class, 96–98
local functions, 90–96
function(s) (continued)

multi-function recursion, 92–93
Neko and C/C++, 568–570
overriding in inheritance, 117–118
recursive functions, 88–90
return from, 86–87
static functions, 114–115
type parameters, use of, 124–125

function keyword, 28

G

games, developing. See Neko Media Engine (NME)
garbage collection
abstract values, 572–573
Neko and C/C++, 571–573
get, hash method, 49
getContent, 123, 307
getDate, 50, 51
gDay, 50, 51
getMonth, 50, 51
getMonthDays( ), 54
getOnlineInfo, 110
getSeconds, 50, 53
GET method, web server, 227–228
getMinutes, 50, 53
getMonth, 50, 51
getMonthDays( ), 54
getTime, 51, 53
Glade, 503–506
downloading, 504
Glade class, 505–506
with hxGtk library, 503–506
new project, starting, 504–505
global cache, SPOD system, 295
global constants, ActionScript/haXe comparison,
346–347
global functions, ActionScript/haXe comparison, 347–352
global properties, ActionScript/haXe comparison, 356

graphical user interface (GUI) development
with Flash. See Flash user interface
with Glade, 503–506
hxGtk library, 499–506
installing libraries, 481
libraries, 369, 480–481
with Neko. See Neko desktop applications
greater than ( > ), 55, 58–59
greater than or equal to ( >= ), 55
GridLayout, Flash user interface, 379–381
grouping operators ( ), 55, 57
groups, regular expressions, 213
GTK. See hxGtk library
Gtk class, 501
gtranslator project, 147

H

hash tables, 48–50
adding/removing items from, 49–50
cryptographic hashes, 224
functions of, 48
hash methods/descriptions, 49
iterating over, 84
keys, use of, 48–49
querying keys in, 49–50

hasNext( ), 81

haXe
abstract value types, 35–36
arrays, 40–45
basic operation of, 8
basic requirements, 9
benefits to use, 5, 7, 21, 73–74
casting, data conversion, 38–40
C/C++ with, 555–575
commenting, 37
compiling to JavaScript, 6–7
compiler directives, 6
compiling, basic operation, 17–19
compiling to Flash, 5–6
compiling to JavaScript, 6–7, 21–22
compiling to Neko, 7, 19–21
conditional statements, 74–79
cross-platform. See regular expressions; Timer
class; XML, Message0–Digest 5 (MD5)
databases, working with. See Neko database support
dates, 50–54
depthbugging, 179–199
file extension (.hx), 8
file extension (.hxml), 17–19
and Flash, 315–385
functional programming support, 7
functions, 85–98
hash tables, 48–50
installation of, 12–14
and JavaScript, 387–423
keywords, 28
language, capabilities of, 7
libraries, 8
lists, 46–48
loops, 79–85
magic, 475–478
-main switch, importance of, 19
math class, 63–66
as object-oriented language, 27, 74
operators, 55–63
program structure, 22–23
and reflection API, 464–469
remoting, 425–452
Runtime Type Information (RTTI), 470–473
serialization, 473–474
simple value types, 31–34
standard data types, 28
string functions, 66–72
switches, 23–25
templates, 251–268
text editor, use with, 17
untyped block, 36–37
variables, 28–31
web development, 230–250
Web site, 9, 12
and XML, 455–464
haxealtdate project, 147
haxedoc
offline documentation, 167–169
syntax for, 167
haxe.Http class, 415
haxelib libraries, 146–149
haxelib tool, use of, 148–149
included projects, listing of, 146–148
installing, 148–149
haxelib-test project, 147
haxe.Log class, 181–185
haxe.Log trace, 181–182
haxe package, 145
haxe.PosInfos, 184
haxe.Template, compared to mtwin.Templo, 261
haxe.unit package. See unit testing
haxORMap project, 147
Hello World
in C/Neko compatible, 556–559
compiling, basic operation, 18–22
creating, 17
here identifier, trace, 182–184
hex, 69
hook values, custom events with, 516–517
HTML, 228–231
and AJAX, 414–422
attribute descriptions, 590–599
attribute groups, 589
attribute value types, 600–601
comments, 588
controls, 585–586
data validation in forms, 404–414
document definition elements, 577–578
Document Object Model (DOM), 392–396
document structure, 229–230
elements in, 226–227
files, root location, changing, 21–22
forms, 585–586
compared to haXe, 230–231
hyperlink format, 583
image inclusion, 583
link elements, 588
lists, 581–582
maps, 584
rules, adding to documents, 405
scripts, 587
structural elements, 578–581
style sheets, 586–587
syntax for, 229, 230
tables, 582–583
table sorting, 401–404
tags in, 226–227
web server, flow in, 226
htmlEscape, 69, 71
htmlUnescape, 69
HTTP protocol, 226
haXe.Http class, 415
JavaScript-to-Neko HTTP server, 433
HTTPS protocol, 226
hxasm project, 147
hxDev project, 147
hxDiff project, 147
hxGtk library, 499–503
events, 502–503
function proxy, 499–501
Glade, use with, 503–506
Gtk class, 501
pros/cons of, 481
HxGtk project, 147
HxJSON project, 147
HxLib project, 147
hxScriptlet project, 147
hxServlet project, 147
hxSynth project, 147
hyperlinks, format of, 583
identifiers, instance fields, sequence of, 107
if statement, 74–77
nesting, 75–76
images
in Flash movie, 323–324
HTML, 583
in Neko Media Engine, 538
implicit import, 141–142
importing
packages, 141–145
templates into templates, 256
increment (++) binary operator, 55, 57–58
index, database fields, accessing by, 284
indexOf, 66–67
inequality (!=), 56, 58–59
inference, function type, 87
info projectname project, 148
Infos interface, magic, 477
inheritance, 115–121
abstract classes/methods, 119–120
constructors, overriding, 118
defined, 115
extending class, steps in, 115–116
functions, overriding, 117–118
and instance variables, 120
roles, different at same time, 120–121
single inheritance, 115
and static fields, 120
super identifiers, 117
toString, 119
insert, arrays

Inspect method, RTTI, 471–473
install projectname project, 148
instance fields, 104–114
    constructors, 108–110
    functions sequence, 108
    identifiers sequence, 107
    inconsistencies, avoiding, 105
    instance functions, 104–105
    optional arguments, 106–107
    variables, 107
    variable value modifiers, 110–114
instance functions, 104–107
    null arguments, 105–106
    optional arguments, 106–107
    syntax for, 104
XML, 207
instance variables, and inheritance, 120
instantiating, arrays, 41
int
    standard data type, 28
    type data conversion, 39
integers, bits allowed, 32–33
interfaces, 121–122
    body, structure of, 121–122
    declaration, syntax for, 121
    uses of, 122
IntHash, 84
IntIter object, 81–82
IntIter operator (...), 82
isEmpty, lists, 47
is method, values, comparing types of, 40
isOnline, blog entries, 110
isSpace, 69
Iterable, typedef, 131–133
iteration
    arrays, 42, 98
    database, with ResultSet, 285–286
    hash method, 49
    over hash tables, 84
    IntIter object, 81–82
    lists, 47, 98
    and loops, 80
    template loop expressions, 254–256
iterator
    method, 82–83
    typedef, 131–133

JavaScript, 387–423
    AJAX, 414–422
    benefits to use, 422
    compiling to, 6–7, 21–22
    cross-browser event system solution, 397–399
    execution errors, redirecting, 392
    external libraries, 159–162
    Flash content, embedding with, 319
    FlashJsConnection, 435
    Flash-to-JavaScript connection, 429–431
    functions, use of, 160
    HTML, traversing with, 404–414
    HTML Document Object Model (DOM), 392–396
    HTML files, necessity of, 21–22
    JavaScript-to-Flash connection, 432
    JavaScript-to-Neko HTTP server, 433
    libraries, 391–392, 399–400
    low-level access to, 396
    magic, 478
    Mozilla compared to Internet Explorer, 7
    Object Notation (JSON), 414
    OnLoad problem, 396–399
    output size, modifying, 390
    Remoting API communication methods, 427–428
    SocketConnection, 436
    switches, 24
    table sorting, 401–404
    trace output, 180
    Unicode support, 221–222
join
    arrays, 42, 68
    lists, 47
jQuery library, 400
js package, 146

K
keyboard, events, Flash user interface, 372–373
keying, color key, Neko Media Engine, 540
keys, hash tables, 48–49
keywords
    ActionScript/haXe comparison, 354–358
    of haXe, 28
kind method, Neko, 304
kother method, Neko, 304

L
labels
    Flash user interface, 376
nGui Label control, 484–485
Lambda class, 96–98
    array, 98
    fold, 98
    map, 97–98
    mapi, 97–98
    methods/description, 97
last, lists, 47
lastIndexOf, 67
lastInsertID method, 281
layout constructors, Flash user interface, 374
left shift (<<), binary operator, 55
length
arrays, 41
database property, 286
lists, 47
string, 66
less than (<), 55, 58–59
less than or equal to (<=), 55
lhx project, 147
libraries, 145–162
external libraries, 150–162
features of, 8
GUI libraries, 480–481
hxLib libraries, 146–149
hxGtk library, 499–503
JavaScript, 391–392, 399–400
Neko, 272–273
from other projects, use of, 150
standard haXe, 145–146
SysTools, 525–530
line commenting, 37
links
dynamic links, 154–156
HTML hyperlinks, 583
HTML link elements, 588
static links, 152–154
Linux
HaXe installation on, 13–14
Neko installation on, 16–17
Ubuntu, Apache installation on, 233–234
list(s), 46–48
adding/removing items from, 48
filter method, 95
functions of, 46–47
HTML, 581–582
iterator, creating, 98
list fields/descriptions, 47
map method, 94–95
querying values in, 48
List control, nGui, 491–495
list method, lambda class, 98
list project, 148
LocalConnection, 435
local functions, 90–96
multi-function recursion, 92–93
passing functions to, 92
structure of, 90
type structure, 91
variable scope, 93–94
Logger class
data requirements, 311–312
Neko, 311–314
stream for output, accepting, 311–312
logical comparison operators, 58–59
logical expressions, 254
Templo, 263
look-around constructs
regular expressions, 220–221
syntax for, 221
loops, 79–85
break and continue, 84–85
for loop, 81–82
looping over collections, 82–84
recursive functions, 88–90
ResultSet class iteration, 285–286
template loop expressions, 254–256
Templo loop expressions, 263–264
while loops, 80–81
Loops, output string data to stream, 313
ipad, 69–70
Lsys project, 147
ltrim, 69–70
M
macros
calling inside template, 256–257
with Templo, 260, 266–268
magic, 475–478
boot classes, 475
Flash, 477–478
Infos interface, 477
JavaScript, 478
Neko, 477
PosInfo, 477
Public interface, 477
—resolve method, 475–477
—setfield method, 475–477
static initialization, 475
main function
execution at startup, 85, 104
HaXe program structure, 23
search, storing location of, 68
-main switch
functions of, 19, 24
importance of, 19
make method, Manager class, extending, 295
Manager class
extending class, steps in, 294–295
methods/description, 292–293
Neko, 288, 291–295
Neko Media Engine (NME), 534–537
nGui library, 482–484
map<X>, lists, 47
mapi method, lambda class, 97–98
map method
Lambda class, 97–98
lists, 94–95
maps, HTML, 584
mask
Flash movie, 342–344
Neko Media Engine (NME), 540

M
macros
calling inside template, 256–257
with Templo, 260, 266–268
magic, 475–478
boot classes, 475
Flash, 477–478
Infos interface, 477
JavaScript, 478
Neko, 477
PosInfo, 477
Public interface, 477
—resolve method, 475–477
—setfield method, 475–477
static initialization, 475
main function
execution at startup, 85, 104
HaXe program structure, 23
search, storing location of, 68
-main switch
functions of, 19, 24
importance of, 19
make method, Manager class, extending, 295
Manager class
extending class, steps in, 294–295
methods/description, 292–293
Neko, 288, 291–295
Neko Media Engine (NME), 534–537
nGui library, 482–484
map<X>, lists, 47
mapi method, lambda class, 97–98
map method
Lambda class, 97–98
lists, 94–95
maps, HTML, 584
mask
Flash movie, 342–344
Neko Media Engine (NME), 540

M
macros
calling inside template, 256–257
with Templo, 260, 266–268
magic, 475–478
boot classes, 475
Flash, 477–478
Infos interface, 477
JavaScript, 478
Neko, 477
PosInfo, 477
Public interface, 477
—resolve method, 475–477
—setfield method, 475–477
static initialization, 475
main function
execution at startup, 85, 104
HaXe program structure, 23
search, storing location of, 68
-main switch
functions of, 19, 24
importance of, 19
make method, Manager class, extending, 295
Manager class
extending class, steps in, 294–295
methods/description, 292–293
Neko, 288, 291–295
Neko Media Engine (NME), 534–537
nGui library, 482–484
map<X>, lists, 47
mapi method, lambda class, 97–98
map method
Lambda class, 97–98
lists, 94–95
maps, HTML, 584
mask
Flash movie, 342–344
Neko Media Engine (NME), 540
math, 63–66
  divide by zero issue, 64–65
  float to int conversion functions, 65–66
  math fields/descriptions, 63–64
  numerical validity, testing for, 65
  template numeric/calculation expressions, 253
menus
  creating, nGui, 496–499
  creating, SysTools, 526–529
  submenus, 497, 528
message boxes
  confirm box, 530
  SysTools, 529–530
message hooks, custom events with, 516–517
methods, abstract methods, 119–120
MochiKit library, 400
mod_neko
  installing on Linux Ubuntu, 233–234
  installing on Windows, 232–233
modulo operator (%), 55, 57
  in Templo, 261
mootools library, 400
Motion-Twin ActionScript Compiler (MTASC), 6
movie clips. See Flash movie; SWFMILL
mtwin project, 147
multidimensional arrays, 45–46
  Arrays of Arrays, 46
multimedia
  Flash movies, 323–324
  with Neko. See Neko Media Engine (NME)
multiplication (*), binary operator, 55
MySQL database, connecting to, 274

N
name qualifier, 353
negation operator (!), 55
if statement, 75
Neko
  applications, running, 20
  basic requirements, 9
  builtins, 477
  bytecode, 156–157
  and C/C++. See Neko and C/C++ compiling to, 7, 19–21
databases, working with. See Neko database support
desktop applications. See Neko desktop applications
downloading, 15
executable files, creating, 20
external libraries, 156–158
Flash-Neko communication, 517–524
front controller, web development, 238–239
functions of, 271–272
installation of, 14–17
integers, bits allowed, 32–33
JavaScript-to-Neko HTTP server, 433
multimedia with. See Neko Media Engine (NME)
ndll library files, 555–556
NekoML, 9
Neko script, standard, 9
NekoSocketConnection, 436–437
NekoTools, 20, 480
neko.vm.Loader, 157
neko.Web class, 236–238
nGui library, 481–498
NXML, 9
Page Controller, web development, 235–236
Remoting API communication methods, 428
Screen Weaver HX application, 511–513
source files, requirements for compiling, 16
  standard libraries, 272–273
switches, 24
Temple system, 258–268
toolkit, 20
trace output, 180
Unicode support, 221–222
virtual machine, 272
web server. See NekoTools web server
Web site, 9, 15
neko_error function, 574
Neko and C/C++
  abstract values, 563–565
databases, 567–568
  C/C++ values, passing to Neko, 562–563
  C foreign function interface (C FFI), 555
data restructuring, 560
error handling, 573–574
functions, Neko, executing in C/C++, 569–570
  functions, passing from C/C++, 569
  functions, passing from haXe, 568–569
garbage collection, 571–573
haXe, data transit from, 559–560
Hello World, 556–559
Neko library files, 556
Neko primitives, 570–571
objects, creating/modifying, 566–567
  objects, passing from Neko, 565–566
type checking, 560–561
value struct, 558–559
Neko database support, 273–314
  choosing database, 273–275
  Connection class, 277–278
database type, determining, 281
data from executed query, container for, 283–286
data integrity, insuring, 281–282
dbName method, 281
deleting records, 280–281
directory management, 303–304
fields, access by index, 284
fields, access by name, 283–284
File class, 304–311
FileSystem class, 301–302
inserting records, 281
iteration, 285–286
lastInsertID method, 281
length property, 286
Logger class, 311–314
Manager class, 291–295
of MySQL database, 274
nfields property, 284–285
object to relational mapping, 286
ODBC driver, 277
path strings, 302–303
populating database, 279–281
of PostgreSQL database, 274
read data directly, 307
read data from stream, 305–307
remote files, accessing, 311
request method, 278
ResultSet class, 283–286
SPoD system. See SPoD (Simple Persistent Objects)
Database) system
SQLite database, 274–275
tables, creating, 278–279
transactions, 281–282
traversing stream, 309–310
updating records, 280
write data to stream, 307–309
Neko desktop applications, 479–506
benefits to use, 479–480
Button controls, 488–489
container classes, 487–488
Control class, 485–487
GUI libraries, 480–481
GUI library installation, 481
Label control, 484–485
List control, 491–495
Manager class, 482–484
menus, 496–499
nGui library controls, 482
static controls, 484–485
submenus, 497
Tab control, 495–496
Text control, 489–491
Neko Media Engine (NME), 532–553
animation, 549
collisions, detecting, 543–545
color key, setting, 540
display buffer, flipping, 535–536
events, capturing, 549–553
images, file formats for, 538
Manager class, 534–537
refresh rate, slowing, 536–537
and SDL library, 532–533
sprites, 546–549
Surface class, 537–545
surface transforms, 540–543
text, adding, 537
timers, 546
Neko.ndlls library, 158
neko package, 146
NekoTools web server, 231–234
deploying website to, 20–21
mod_neko, installation of, 232–234
parameters, listing of, 231
root location, changing, 21, 232
TicTacToe Game, 443–452
Neko Virtual Machine, 480, 509
neko.Web class, 236–238
static methods, 237–238
static variables, 238
nesting, if statement, 75–76
newlines, escaped characters, 33–34
next( ), 81
nfields property, database, 284–285
nGame project, 147
nGui library. See Neko desktop applications
nMagick project, 147
nme project, 147
nodes, XML
checking, 458–461
creating, 206
traversing, 456–457
–no-traces compiler directive, 185
nPostgres project, 147
null, variable value modifier, 111
Null<T>, 345
null arguments, instance functions, 105–106
null type, functions of, 35
numbers, floating-point, 32
numeric expressions, 253

0
object(s), anonymous objects, 126–127
object initializer, 353
Objective Caml, downloading, 13
object-oriented programming
anonymous objects, 126–127
classes, 102–103, 122–126
constructor arguments, 136–138
dynamic type, 127–133
elem, 134–136
extensions, 133–134
with haXe, 27, 74
inheritance, 115–121
instance fields, 104–114
interfaces, 121–122
polymorphism, 118
static fields, 114–115
object types, Neko and C/C++

- ODBC driver, Neko database connection, 277
- onClose events, 515
- onFilesDropped events, 516
- online documentation, 169–170
- onMinimize/onMaximize events, 515
- onRightClick events, 516
- opengl project, 147
- operators, 55–63
  - ActionScript/haXe comparison, 352–353
  - assignment operators, 59–60
  - binary operators, 56–58
  - bitwise operators, 60–63
  - listing of, 55–56
  - logical comparison operators, 58–59
- optional arguments, 106–107
--placement of, 106
- OR ( | ), bitwise operator, 62
- OR ( | | ), logical comparison operator, 56, 59

P
- packages, 139–145
  - declaring package, 140–141
  - and enum, 143
  - explicit import, 142–143
  - implicit import, 141–142
  - importing package, 144–145
  - type lookup sequence, 144
- Page Controller, Neko as, 235–236
- PanelLayout, Flash user interface, 376–378
- parseFloat, type data conversion, 39
- parseInt, type data conversion, 39
- pascal4neko project, 147
- passwords, and cryptographic hashes, 224
- path projectname project, 148
- path strings, Neko, 302–303
- patterns, regular expressions, 214–222
- Person class, 438
- PersonDatabase class, 438–440
- polymorphism, 118
- pop, arrays, 42, 43
- PosInfo, magic, 477
- PostgreSQL database, connecting to, 274
- POST method, web server, 227–228
- primitives, Neko and C/C++, 570–571
- PRIVATE_FIELDS, 290
- private classes, 102
- properties, ActionScript/haXe comparison, 356
- Prototype library, 400
- proxy
  - class, remoting, 437–442
  - hxGtk function proxy, 499–501
  - XML Proxy, 463–464
- public classes, 102
- Public interface, magic, 477

Q
- quantifiers
  - regular expressions, 218–219
  - syntax for, 218
- queued actions, timer, 223

R
- raw expression, Templo, 262–263
- RChoice, 460
- RData, 460–461
- readAll method, 306–307
- readBytes method, 306
- readChar method, 305–306
- readDirectory method, Neko, 303–304
- read-only access, variable value modifier, 111–112
- recursive functions, 88–90
  - functions of, 88
  - multifunction recursion, 92–93
  - operation of, 89–90
- Reflect class, 122–123
- reflection API, 464–469
  - Reflect class, 464–466
  - Type class, 467–469
- refresh rate, slowing, 536–537
- register project, 149
- regular expressions, 210–222
  - alternation, 220
  - anchors, 217–218
  - back-references, 214, 220
  - character classes, 215–217
  - characters, 214–215
  - character sets, 217
  - conditionals and comments, 220–221
  - defined, 210
  - EReg class, 210–214
  - groups, 219
  - look-around constructs, 220–221
  - optional modifiers, 211–214
  - patterns, 214–222
  - quantifiers, 218–219
  - replace method, 213–214
  - split method, 213
  - Unicode support, 221–222
- RELATIONS method, SPOD system, 297–298
remoting, 425–452
AsyncAdapter, 437
asyncConnection, creating, 432–434
AsyncDebugConnection, 437
connection, creating, 428–429
DelayedConnection, 437
FlashJsConnection, 435
Flash-Neko communication, 517–524
Flash-to-Action Message Format (AMF) Server connection, 434
Flash-to-JavaScript connection, 429–431
functions of, 426
JavaScript-to-Flash connection, 432
JavaScript-to-Neko HTTP server, 433
LocalConnection, 435
NekoSocketConnection, 436–437
proxy class, 437–442
push or pull type communication, 426–427
remote files, accessing, Neko, 311
Remoting API communication methods, 427–428
server side requirements, 426
SocketConnection, 436
synchronous versus asynchronous communication, 426
TicTacToe Game, 443–452
remove
arrays, 42, 44
hash method, 49
lists, 47
remove projectname project, 149
replace
characters in string, 71
regular expressions method, 213–214
StringTools method, 69
request method, Neko database support, 278
reserved characters, regular expressions, 215
—resolve method, magic, 475–477
resources, 163–166
—embedded XML files, use of, 163
resourcefile path, 163
templates, 257–258
ResultSet class, 283–286
return keyword, end of function, 86–87
right shift (>>,), binary operator, 55
RList, 459–460
RMulti, 459
RNode, 459
ROptional, 460
round, float to int conversion, 66
rpad, 69–70
rtrim, 69–70
rules, HTML forms, data validation, 404–414
run projectname project, 149
Runtime Type Information (RTTI), 470–473
—accessing with haXe, 471
function of, 470
Inspect method, 471–473
XML RTTI documents processing, 470–473
S
Screen Weaver HX, 510–524
ActionScript extensions, 523–524
benefits to use, 510, 513
downloading, 511
with Flash. See Flash and Screen Weaver HX
Flash-Neko communication, 517–524
installing, 511
library, 480
menus, creating, 526–529
message boxes, creating, 529–530
Neko application, example of, 511–513
SWF2Exe applications, 510
synchronous communication system, 520–524
system tray icon, creating, 526
SysTools library, 525–530
window events, 515–516
window flags, 513
window properties, 514
script.aculo.us library, 400
scripts, HTML, 587
search
—storing location of, 68
strings, 67
search keyword project, 149
SeekBegin method, 310
SeekCur method, 310
SeekEnd method, 310
seek method, 309–310
serialization, 473–474
—defined, 473
—serialization API, 473–474
—serialize/deserialize process, 474
set, hash method, 49–50
setColor function, trace, 184
set expression, Templo, 264–265
—setter method, variable value modifier, 111
setup project, 149
shifting bits, 61–62
simple value types, 31–34
—Booleans, 33
—floating-point numbers, 32
—strings, 33–34
single inheritance, 115
single quotes, escaped characters, 34
slice, arrays, 42, 44–45
slideshow, Flash, 331–337
socket connections
—NekoSocketConnection, 436–437
—SocketConnection, 436
sort method, arrays, 96
sounds, Flash movie, 325
spaces, in strings, removing, 70
special values, ActionScript/haXe comparison

splice, arrays, 42, 44
split, 66–67
split method, regular expressions, 213
SPOD (Simple Persistent Objects Database) system, 286–301
  Author class, 287–288, 296–297, 299
  AuthorManager class, 300–301
  Chapter class, 296–297, 299–300
  ChapterManager class, 300–301
  deleting records, 291
  functions of, 286
  global cache, 295
  inserting data, 291
  RELATIONS method, 297–298
  synchronize data, 291
  table mapping, 289–291
  table relations, 295–301
  updating records, 291
sprites
  animating, 548
  Neko Media Engine (NME), 546–549
  sprite sheet, 547–548
SQLite database
  connecting to, 274–275
  hxGtk library, use of, 499–500
  pros/cons of, 275
StackLayout, Flash user interface, 378–379
stage, Flash movie, 316–317
standard data types
  haXe types, 28
  simple value types, 31–34
startsWith, 70
statements, ActionScript/haXe comparison, 354–358
static controls, nGui, 484–485
static fields, 114–115
  functions of, 114
  and inheritance, 120
  static functions, 114–115
  static variables, 114–115
static fromCharCode, 66
static fromString, date object, 51
static functions, XML, 206
static initialization, magic, 475
static links, Flash, 152–154
static methods, neko.Web class, 237–238
static now, date object, 51
static variables, 103, 114–115
  declaring, syntax for, 114
  neko.Web class, 238
  values, assigning, 115
static fromTime, date object, 51
streams
  output string data to, 313
  read data from, 305–307
  stream for output, accepting, 311–312
  traversing, 309–310
  write data to, 307–309
string(s), 66–72
  add characters to end of, 70
  arrays converted to, 68
  characters, extracting from, 67
  constructing from smaller strings/characters, 71–72
  converting to arrays, 67
  delimiters, 67
  encoding for web, 71
  escaped characters, 33–34
  fields/descriptions, 66
  functions of, 33
  length, determining, 67
  replace characters, 71
  searching strings, 67
  starting/ending characters, verifying, 70
  StringBuffer method, 71–72
  string literals, 33–34
  StringTools methods, 69–70
  unwanted spaces in, 70
  verify start and end of, 70
string, standard data type, 28
string, type data conversion, 39
submenus
  nGui, 497
  Screen Weaver HX, 528
submit file.zip project, 149
substr, 66–67
subtraction (-), binary operator, 55
super identifiers, 117–118
Surface class, Neko Media Engine (NME), 537–545
SWFMill, 364–369
  file formats supported, 365
  functions of, 364
  movie clip, creating, 365
  movie clip use with haXe, 365–368
  parameters/description, 364
  transition between frames, 367–368
  True Type Fonts (TTF), embedding, 368–369
SWF Studio, 510
swhx project, 147
switch(es)
  all platforms, 24–25
  Flash, 24
  JavaScript, 24
  -main switch, importance of, 19
  Neko, 24
switch statement, 77–78
  and enum, 135, 143
  values, returning from conditional statements, 78–79
synchronous communications, 426
in ActionScript, 523–524
in haXe, 520–523
Screen Weaver HX system, 520–524
SysTools
   features, listing of, 525
   menus, creating, 526–529
   message boxes, 529–530
   Screen Weaver HX, 525–530
   tray icon, creating, 526
systools project, 147

T
Tab control, nGui, 495–496
table(s)
   database, creating, 278–279
   HTML, 582–583
   many-to-many relations, creating, 298–301
   mapping, SPOD system, 289–291, 296
   relations, adding, SPOD system, 295–301
   sorting, HTML, 401–404
TABLE_IDS, 290
TABLE_NAME, 290
tabs, escaped characters, 33–34
tags, HTML, 226–227
TDD (Test-Driven Development), 172–177
tell method, 309
templates, 251–268
   class methods/description, 252
   code, situations for use, 258
   comparison expressions, 253
   dot expressions, 253–254
   file extensions, 260
   functions of, 251–252
   importing into templates, 256
   logical expressions, 254
   loop expressions, 254–256
   macros, use of, 256–257
   manual compilation of, 268
   numeric/calculation expressions, 253
   resources, 257–258
   syntax for, 253–256
   Templo, template engine, 258–268
Templo, 258–268
   attr expressions, 262
   features of, 258–259
   file extensions, 260
   fill expression, 265
   haxe.Template compared to mtwin.Templo, 261
   installing, 259
   logical expressions, 263
   loop expressions, 263–264
   macros, 260, 266–268
   raw expression, 262–263
set expression, 264–265
Template system, 259–260
use expression, 265–266
TestCase, unit testing, 171–172
test file.zip project, 149
text
   box, Flash user interface, 376
   editor, use with haXe, 17
   in Flash movie, 322–323
   Flash user interface, 375
   nGui Text control, 489–491
   True Type Fonts, embedding in SWFMill movie clips, 368–369
   writing, Neko Media Engine (NME), 537
TextField class, Flash, 322–323
this keyword, current object, calling, 104
throwing exceptions, 188
TicTacToe Game, 443–452
time. See date/time; timer
timeline, Flash movie, 316–317
timer, 222–223
   action, delayed, 223
   functions of, 222
   Neko Media Engine (NME), 546
   query time, 546
   queued actions, 223
   timer instance, creating, 222
token, date formats, 54
toLowerCase, 66
tools package, 146
toUpperCase, 66
trace, 179–185
clear function, 184
Flash output, 180
haXe compared to ActionScript, 181
haXe.Log class, 181–185
haxe.Log.trace, 181–182
haxe.PosInfos, 184
here identifier, 182–184
JavaScript output, 180
Neko output, 180
--no-traces compiler directive, 185
redirect to Firebug, 389–390
setColor function, 184
transactions, database, 281–282
transforms, surface, Neko Media Engine, 540–543
tray icon, creating with SysTools, 526
trim, 70
true, and if statement

true, and if statement, 75–76
True Type Fonts (TTF), embedding in SWFMII
movie clips, 368–369
try...catch block, 304–305
twister project, 148
type(s), lookup sequence, 144
type checking, Neko and C/C++, 560–561
Type class, 122–123
reflection API, 467–469
typedef, 128–131
with anonymous objects, 128–130
benefits to use, 129–130
declaring, syntax for, 128
extensions, use of, 134
external structure mapping, 150
HTML DOM declarations, 394–396
Iterator and Iterable, 131–133
keyword, 28
type operator, 353
type parameters, 123–126
constraints on, 125–126
at function level, 124–125
situations for use, 123–124
typing
dynamic typing, 29
type inference, 30–31
untyped block, 36–37

U
Unicode support
characters, embedding in source code, 222
scope of, 221–222
unit testing, 170–177
haxe.unit package, 171–177
purpose of, 170–171
TDD (Test-Driven Development), 172–177
TestCase method, 171–172
unknown keyword, 28
unmake method, Manager class, extending, 295
Unobtrusive Flash Objects (UFO), Flash content, embedding with, 319
unpublish, blog entry, 109–110
unsafe cast, 38–39
Unserializer class, 474
unshift, arrays, 42, 43
untyped block, 36–37
upgrade project, 149
URI (Unified Resource Identifier)
URL rewritten to, 231
web server, 226, 228
urlDecode, 70
urlEncode, 70, 71
use expression, Templo, 265–266
user interface. See graphical user interface (GUI) development
user username project, 149

V
val_print function, 574
val_throw and val_rethrow function, 573
value(s)
arrays, adding/removing, 43–44
C/C++ values, passing to Neko, 562–563
comparing types of, 40
Flash movie, loading, 325
lists, adding/removing, 47
querying in lists, 48
returned by functions, 87
returning from conditional statements, 78–79
variable modifiers of, 110–114
value struct, 558–559
variables, 28–31
of classes, 102–103
constant variables, 31
declarations, 29–30
dynamic typing, 29
Flash movie, loading, 326–327
identifiers sequence, 107
initializing, 29
instance fields, 107
local functions, 90–91, 93–94
modified, syntax for, 110
scope, local functions, 93–94
static variables, 103, 114–115, 238
type inference, 30–31
value modifiers, 110–114
var keyword, 29
Window class, 514
XML, 208
var keyword, 29
vector graphics, Flash drawing API, 337–342
VectorSpace project, 148
virtual machine, Neko, 272
void
functions of, 35
standard data type, 28, 87

W
Web Developer extension, FireFox, 389
web development, 225–250
Apache web server, 232–234
behavior layer problems, 388–389
blog entries, 108–110
Firefox extensions, 389
GUI. See graphical user interface (GUI) development
haxe compared to HTML, 230–231
haxe.Http class, 415
HTML, 228–231
movie clips. See Flash; Flash movies
remote communication. See remoting
Screen Weaver HX, 510–524
Index

site domains, levels of, 227
strings, encoding for web, 71
web page, layers of, 387–388
web server, 226–228
Wiki site development example, 238–250
See also Java Script; Neko

web server, 226–228
accessing, addresses for, 227
content distribution with, 226–228
functions of, 226
GET method, 227–228
information flow in, 226
NekoTools. See NekoTools web server
POST method, 227–228
Remoting API communication methods, 427–428
templates, Templo, 258–268
URI addresses, 226, 228

websites, development of. See web development

while loops, 80–81
break and continue with, 84–85
while and do while, 80–81

Wiki, site development, example of, 238–239

Windows
Apache, installation of, 233–234
HaXe installation on, 12–13
Neko installation on, 14–16
Screen Weaver HX functionality, 513–517
Unicode support, 221–222
writeBytes method, 308–309

X
xcross project, 148
Xinf, 480
xinf project, 148
XML, 455–464
attributes, accessing, 457–458
attributes, checking, 461–462
class instance functions, 207
class static functions, 206
class variables, 208
config information, storing, 163–165
document creation, 209–210
documents, structure of, 203–206
Fast class, 455–456
filter rules, 462–463
Flash movie, loading, 327–328
haXe API, 206–208
nodes, checking, 458–461
nodes, creating, 206
nodes, traversing, 455–457
online documentation, 169–170
Proxy class, 463–464
RChoice, 460
RData, 460–461
RList, 459–460
RMulti, 459
RNode, 459
ROptional, 460
SWFMill movie clips, 364–368
traversing documents, 208–209
validation rule, 458
XML RTTI documents processing, 470–473
XOR (^), bitwise operator, 56, 62
xpath project, 148

Y
Yahoo! User Interface Library, 400

Z
zero, divide by zero issue, 64–65
Zinc, 510
Now you can access more than 200 complete Wrox books online, wherever you happen to be! Every diagram, description, screen capture, and code sample is available with your subscription to the Wrox Reference Library. For answers when and where you need them, go to wrox.books24x7.com and subscribe today!