

Shay Friedman

# IronRuby

**UNLEASHED**



**SAMS**

Shay Friedman

# IronRuby

**UNLEASHED**



800 East 96th Street, Indianapolis, Indiana 46240 USA

## IronRuby Unleashed

Copyright © 2010 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-672-33078-0

ISBN-10: 0-672-33078-4

*Library of Congress Cataloging-in-Publication Data*

Friedman, Shay.

IronRuby unleashed / Shay Friedman.

p. cm.

ISBN 978-0-672-33078-0

1. IronRuby (Computer program language) 2. Microsoft .NET Framework. 3. Ruby (Computer program language) I. Title.

QA76.73.I586F74 2010

006.7'882—dc22

2009050114

Printed in the United States of America

First Printing: February 2010

## Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Pearson Education, Inc. cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

## Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

## Bulk Sales

Pearson offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact:

**U.S. Corporate and Government Sales**

**1-800-382-3419**

**corpsales@pearsontechgroup.com**

For sales outside of the U.S., please contact:

**International Sales**

**+1-317-581-3793**

**international@pearsontechgroup.com**

## Editor-in-Chief

Karen Gettman

## Executive Editor

Neil Rowe

## Acquisitions Editor

Brook Farling

## Development Editor

Mark Renfrow

## Managing Editor

Kristy Hart

## Project Editor

Andy Beaster

## Copy Editor

Keith Cline

## Indexer

Word Wise Publishing  
Services

## Proofreader

San Dee Phillips

## Technical Editor

Justin Etheredge

## Publishing

### Coordinator

Cindy Teeters

## Interior Designer

Gary Adair

## Cover Designer

Gary Adair

## Compositor

Nonie Ratcliff

# Contents at a Glance

|   |     |
|---|-----|
| Introduction .....  | 1   |
| <b>Part I Introduction to IronRuby</b>                            |     |
| <b>1</b> Introduction to the Ruby Language .....                  | 5   |
| <b>2</b> Introduction to the .NET Framework .....                 | 13  |
| <b>3</b> Introduction to the Dynamic Language Runtime (DLR) ..... | 21  |
| <b>4</b> Getting Started with IronRuby .....                      | 25  |
| <b>Part II The Ruby Language</b>                                  |     |
| <b>5</b> The Basic Basics .....                                   | 43  |
| <b>6</b> Ruby's Code-Containing Structures .....                  | 87  |
| <b>7</b> The Standard Library .....                               | 131 |
| <b>8</b> Advanced Ruby .....                                      | 161 |
| <b>Part III IronRuby Fundamentals</b>                             |     |
| <b>9</b> .NET Interoperability Fundamentals .....                 | 207 |
| <b>10</b> Object-Oriented .NET in IronRuby .....                  | 239 |
| <b>Part IV IronRuby and the .NET World</b>                        |     |
| <b>11</b> Data Access .....                                       | 259 |
| <b>12</b> Windows Forms .....                                     | 281 |
| <b>13</b> Windows Presentation Foundation (WPF) .....             | 303 |
| <b>14</b> Ruby on Rails .....                                     | 331 |
| <b>15</b> ASP.NET MVC .....                                       | 363 |
| <b>16</b> Silverlight .....                                       | 401 |
| <b>17</b> Unit Testing .....                                      | 425 |
| <b>18</b> Using IronRuby from C#/VB.NET .....                     | 459 |
| <b>Part V Advanced IronRuby</b>                                   |     |
| <b>19</b> Extending IronRuby .....                                | 477 |
| Index .....   | 511 |

# Table of Contents

|   |           |
|---|-----------|
| <b>Introduction</b>                                   | <b>1</b>  |
| <b>Part I Introduction to IronRuby</b>                |           |
| <b>1 Introduction to the Ruby Language</b>            | <b>5</b>  |
| History of the Ruby Language .....                    | 5         |
| Implementations .....                                 | 6         |
| Features .....  | 6         |
| Dynamic Language .....                                | 6         |
| Object Oriented .....                                 | 7         |
| Duck Typing .....                                     | 8         |
| Metaprogramming .....                                 | 9         |
| REPL .....  | 10        |
| Available Libraries .....                             | 11        |
| <b>2 Introduction to the .NET Framework</b>           | <b>13</b> |
| History of the .NET Framework .....                   | 13        |
| Overview .....  | 15        |
| Features .....  | 16        |
| Common Language Infrastructure .....                  | 16        |
| Assemblies .....                                      | 18        |
| Base Class Library .....                              | 19        |
| Security .....  | 19        |
| Memory Management .....                               | 19        |
| Frameworks .....                                      | 20        |
| <b>3 Introduction to the Dynamic Language Runtime</b> | <b>21</b> |
| Overview .....  | 22        |
| Features .....  | 23        |
| Common Hosting Model .....                            | 23        |
| Runtime Components .....                              | 23        |
| Language Implementation .....                         | 24        |

|                |   |           |
|----------------|---|-----------|
| <b>4</b>       | <b>Getting Started with IronRuby</b>    | <b>25</b> |
|                | Overview .....                          | 25        |
|                | Installing IronRuby .....               | 26        |
|                | IronRuby Folders .....                  | 29        |
|                | Getting the Sources .....               | 29        |
|                | Executables and Tools .....             | 30        |
|                | The IronRuby Interpreter (ir.exe) ..... | 31        |
|                | File Execution Mode .....               | 32        |
|                | Development Environments .....          | 34        |
|                | Ruby in Steel .....                     | 34        |
|                | NetBeans .....                          | 35        |
|                | RubyMine .....                          | 36        |
|                | Others .....                            | 37        |
|                | The Power of IronRuby .....             | 38        |
| <br>           |   |           |
| <b>Part II</b> | <b>The Ruby Language</b>                |           |
| <br>           |   |           |
| <b>5</b>       | <b>Ruby Basics</b>                      | <b>43</b> |
|                | Basic Syntax .....                      | 43        |
|                | Comments .....                          | 43        |
|                | Setting Values to Variables .....       | 44        |
|                | Calling Methods .....                   | 45        |
|                | Code File Structure .....               | 46        |
|                | Coding Standards .....                  | 47        |
|                | Hello World .....                       | 48        |
|                | Variables .....                         | 48        |
|                | Numbers .....                           | 48        |
|                | Text .....                              | 50        |
|                | Arrays .....                            | 54        |
|                | Hashes .....                            | 57        |
|                | Ranges .....                            | 59        |
|                | Booleans .....                          | 60        |
|                | Regular Expressions .....               | 60        |
|                | Date and Time .....                     | 62        |
|                | Constants .....                         | 63        |
|                | Control Structures .....                | 64        |
|                | Conditions .....                        | 64        |
|                | Loops .....                             | 70        |
|                | The yield Statement .....               | 76        |
|                | BEGIN and END .....                     | 77        |

|                             |    |
|-----------------------------|----|
| Exception Handling .....    | 78 |
| Exception Information ..... | 78 |
| rescue .....                | 78 |
| else .....                  | 81 |
| ensure .....                | 82 |
| raise .....                 | 83 |
| Custom Error Classes .....  | 85 |

## **6 Ruby's Code-Containing Structures 87**

|  |     |
|--|-----|
| Methods .....  | 87  |
| Defining Methods .....   | 88  |
| Method Naming .....  | 90  |
| Returning a Value from Methods .....                           | 90  |
| Method Name Aliasing .....                                     | 91  |
| Default Parameter Values .....                                 | 92  |
| Special Parameter Types .....                                  | 93  |
| Associate Methods with Objects .....                           | 94  |
| Removing Method Definitions .....                              | 95  |
| Blocks, Procs, and Lambdas .....                               | 96  |
| Blocks .....   | 96  |
| Procs .....  | 97  |
| Lambdas .....  | 99  |
| Flow-Altering Keywords Within Blocks, Procs, and Lambdas ..... | 100 |
| Classes .....  | 101 |
| Defining Classes .....   | 101 |
| Creating a Class Instance .....                                | 102 |
| Defining a Constructor .....                                   | 102 |
| Variables Inside Classes .....                                 | 102 |
| Accessors .....  | 107 |
| Methods .....  | 109 |
| Operator Overloading .....                                     | 111 |
| Special Methods .....  | 115 |
| The self Keyword .....   | 118 |
| Visibility Control .....                                       | 118 |
| Inheritance .....  | 120 |
| Duck Typing .....  | 124 |
| Modules .....  | 126 |
| Module-Contained Objects .....                                 | 126 |
| Namespaces .....   | 127 |
| Mixins .....   | 128 |

|          |                                       |            |
|----------|---------------------------------------|------------|
| <b>7</b> | <b>The Standard Library</b>           | <b>131</b> |
|          | Using the Libraries .....             | 131        |
|          | Libraries Available in IronRuby ..... | 132        |
|          | Libraries Reference .....             | 135        |
|          | Abbrev .....                          | 135        |
|          | Base64 .....                          | 135        |
|          | Benchmark .....                       | 136        |
|          | BigDecimal .....                      | 136        |
|          | Complex .....                         | 137        |
|          | CSV .....                             | 137        |
|          | Digest .....                          | 138        |
|          | E2MMAP .....                          | 139        |
|          | English .....                         | 140        |
|          | Erb .....                             | 141        |
|          | FileUtils .....                       | 143        |
|          | Logger .....                          | 143        |
|          | Monitor .....                         | 144        |
|          | Net/http .....                        | 144        |
|          | Observer .....                        | 145        |
|          | Open-uri .....                        | 145        |
|          | Ping .....                            | 147        |
|          | Rational .....                        | 152        |
|          | Rexml .....                           | 153        |
|          | Singleton .....                       | 154        |
|          | Socket .....                          | 154        |
|          | Thread .....                          | 157        |
|          | YAML .....                            | 157        |
|          | WEBrick .....                         | 157        |
|          | Zlib .....                            | 158        |
|          | Finding More Libraries .....          | 159        |
| <b>8</b> | <b>Advanced Ruby</b>                  | <b>161</b> |
|          | Threads .....                         | 161        |
|          | Exceptions Within Threads .....       | 163        |
|          | Passing Data In and Out .....         | 164        |
|          | Thread Priority .....                 | 164        |
|          | Thread State .....                    | 165        |
|          | Thread Synchronization .....          | 167        |
|          | Handling Files .....                  | 169        |
|          | Reading Files .....                   | 170        |
|          | Writing Files .....                   | 172        |



|  |     |
|--|-----|
| Accessing File Properties .....                    | 173 |
| Listing Directories .....                          | 174 |
| File Operations .....                              | 175 |
| Reflection .....                                   | 176 |
| Finding Living Objects .....                       | 176 |
| Investigating Objects .....                        | 177 |
| Invoke Methods and Set Variables Dynamically ..... | 178 |
| Execute Code Dynamically .....                     | 180 |
| Marshaling .....                                   | 181 |
| Binary Marshaling .....                            | 181 |
| Textual Marshaling .....                           | 182 |
| RubyGems .....                                     | 183 |
| Installing RubyGems .....                          | 183 |
| Installing Gems .....                              | 183 |
| Using Installed Gems .....                         | 183 |
| Rake .....   | 184 |
| IronRuby RubyGems Limitations and Expertise .....  | 185 |
| Finding Gems .....                                 | 185 |
| Design Patterns .....                              | 186 |
| The Strategy Pattern .....                         | 186 |
| The Iterator Pattern .....                         | 188 |
| The Command Pattern .....                          | 190 |
| The Singleton Pattern .....                        | 192 |
| The Observer Pattern .....                         | 194 |
| The Builder Pattern .....                          | 196 |
| Domain-Specific Languages .....                    | 199 |

## Part III IronRuby Fundamentals

|   |            |
|---|------------|
| <b>9 .NET Interoperability Fundamentals</b> ..... | <b>207</b> |
| Bringing .NET into Ruby .....                     | 207        |
| require .....                                     | 207        |
| load_assembly .....                               | 209        |
| load .....  | 210        |
| The \$LOAD_PATH Variable .....                    | 210        |
| .NET Code Mapping .....                           | 210        |
| Types Differences .....                           | 211        |
| Coding Standards Collision .....                  | 211        |
| Private Binding Mode .....                        | 213        |
| Using .NET Objects .....                          | 214        |
| Namespaces .....                                  | 214        |
| Interfaces .....                                  | 216        |

|  |            |
|--|------------|
| Classes .....                                    | 216        |
| Structs .....                                    | 217        |
| Delegates .....                                  | 217        |
| Events .....                                     | 218        |
| Enums .....                                      | 221        |
| Constants .....                                  | 222        |
| Methods .....                                    | 222        |
| Fields .....                                     | 228        |
| Properties .....                                 | 228        |
| Generics .....                                   | 229        |
| Special IronRuby Methods .....                   | 231        |
| Object Class Methods .....                       | 231        |
| Class Class Methods .....                        | 232        |
| Method Class Methods .....                       | 233        |
| String Class Methods .....                       | 234        |
| The IronRuby Class .....                         | 235        |
| CLR Objects and Ruby's Reflection .....          | 237        |
| The Basic Object .....                           | 237        |
| <b>10 Object-Oriented .NET in IronRuby</b> ..... | <b>239</b> |
| Inheriting from CLR Classes .....                | 239        |
| Regular Classes .....                            | 239        |
| Abstract Classes .....                           | 242        |
| Sealed and Static Classes .....                  | 243        |
| Inheriting from CLR Structs .....                | 243        |
| Inheriting from CLR Interfaces .....             | 243        |
| Overriding Methods .....                         | 245        |
| Virtual Methods .....                            | 245        |
| Abstract Methods .....                           | 246        |
| Regular Methods .....                            | 247        |
| Static Methods .....                             | 248        |
| Methods with Multiple Overloads .....            | 249        |
| Sealed Methods .....                             | 250        |
| Overriding Properties .....                      | 251        |
| Overriding Events .....                          | 253        |
| Opening CLR Classes .....                        | 254        |
| Using Mixins .....                               | 254        |
| Opening the Object Class .....                   | 255        |
| Opening Namespaces .....                         | 256        |

**Part IV IronRuby and the .NET World**

|           |  |            |
|-----------|--|------------|
| <b>11</b> | <b>Data Access</b>                             | <b>259</b> |
|           | Hello, Data Access .....                       | 259        |
|           | Preparing Your Environment .....               | 260        |
|           | Contacting a SQL Server .....                  | 260        |
|           | Loading the Needed Assemblies .....            | 260        |
|           | Building the Class Structure .....             | 260        |
|           | Building the Connection String .....           | 261        |
|           | Opening a Connection to the SQL Server .....   | 262        |
|           | Querying the Database .....                    | 263        |
|           | Wrapping Up sql.rb .....                       | 264        |
|           | Using the SqlServerAccessor Class .....        | 265        |
|           | Contacting a MySQL Server .....                | 265        |
|           | Preparing the MySQL Database .....             | 266        |
|           | Loading the Assemblies .....                   | 267        |
|           | Building the Class Structure .....             | 267        |
|           | Building the Connection String .....           | 267        |
|           | Opening a Connection to the MySQL Server ..... | 268        |
|           | Querying the Database .....                    | 268        |
|           | Inserting Records .....                        | 269        |
|           | Deleting Records .....                         | 269        |
|           | Wrapping Up mysql.rb .....                     | 270        |
|           | Using the MySQLAccessor Class .....            | 272        |
|           | Design Considerations .....                    | 272        |
|           | The CachedDataAccess Class .....               | 276        |
|           | Wrapping Up cached_data_access.rb .....        | 277        |
|           | Using the CachedDataAccess Class .....         | 278        |
|           | A Word About LINQ .....                        | 279        |
| <b>12</b> | <b>Windows Forms</b>                           | <b>281</b> |
|           | Introduction .....                             | 281        |
|           | The Application Structure .....                | 282        |
|           | Building the Chat Class .....                  | 282        |
|           | Requiring the Needed Assemblies .....          | 282        |
|           | Initiating the Class .....                     | 282        |
|           | Receiving Messages .....                       | 283        |
|           | Sending Messages .....                         | 283        |
|           | Wrapping Up the Chat Class (chat.rb) .....     | 284        |
|           | Building the Chat Windows Form .....           | 285        |
|           | Loading the Needed Assemblies .....            | 285        |
|           | Building the Class .....                       | 285        |

|   |            |
|---|------------|
| Initializing the Form .....                     | 286        |
| Setting the Form Properties .....               | 287        |
| Adding Controls .....                           | 289        |
| Adding Functionality .....                      | 293        |
| Using the Visual Studio Visual Designer .....   | 295        |
| Wrapping Up the ChatForm Class .....            | 297        |
| Writing the Execution Code .....                | 300        |
| <b>13 Windows Presentation Foundation (WPF)</b> | <b>303</b> |
| Hello, WPF .....                                | 303        |
| XAML .....                                      | 305        |
| Namespaces .....                                | 306        |
| IronRuby and WPF Fundamentals .....             | 307        |
| Running XAML .....                              | 307        |
| Retrieving WPF Elements .....                   | 308        |
| Event Handling .....                            | 308        |
| Windows .....                                   | 309        |
| Window .....                                    | 309        |
| Navigation Window .....                         | 314        |
| Layout Controls .....                           | 317        |
| StackPanel .....                                | 317        |
| Grid .....                                      | 319        |
| Canvas .....                                    | 320        |
| More Panels .....                               | 321        |
| Graphics and Animations .....                   | 321        |
| Shapes .....                                    | 322        |
| Brushes .....                                   | 322        |
| Animations .....                                | 324        |
| Data Binding .....                              | 325        |
| Binding to Static Data .....                    | 325        |
| Binding to Dynamic Data .....                   | 327        |
| REPL .....                                      | 329        |
| <b>14 Ruby on Rails</b>                         | <b>331</b> |
| Preparing Your Environment .....                | 331        |
| Hello, IronRuby on Rails .....                  | 332        |
| Creating the Initial Project Files .....        | 333        |
| Directory Structure .....                       | 333        |
| Database Configuration .....                    | 334        |
| Running the Server .....                        | 337        |

|   |            |
|---|------------|
| The Basic Concepts .....                            | 339        |
| MVC .....   | 339        |
| REST .....  | 339        |
| CoC .....   | 340        |
| DRY .....   | 340        |
| Main Components .....                               | 340        |
| The Model: ActiveRecord .....                       | 340        |
| The View: ActionView .....                          | 341        |
| The Controller: ActionController .....              | 341        |
| Routes .....  | 341        |
| Know Your Environment .....                         | 342        |
| script/server .....                                 | 342        |
| script/generate .....                               | 343        |
| db:migrate .....                                    | 345        |
| Creating a Page .....                               | 346        |
| Generating the Page Controller and View .....       | 346        |
| Helper Classes .....                                | 349        |
| Adding Stylesheets .....                            | 350        |
| Adding Layouts .....                                | 351        |
| Adding Functionality .....                          | 353        |
| Creating a Database-Driven Page .....               | 354        |
| Generating the Page Resources .....                 | 354        |
| Polishing the Index Page .....                      | 356        |
| <b>15 ASP.NET MVC .....</b>                         | <b>363</b> |
| Preparing Your Environment .....                    | 363        |
| Installing ASP.NET MVC .....                        | 364        |
| Obtaining the IronRubyMvc DLL .....                 | 364        |
| Adding IronRubyMvc Templates to Visual Studio ..... | 365        |
| Hello, ASP.NET MVC .....                            | 365        |
| Generating the Initial Project .....                | 365        |
| MVC .....   | 368        |
| Models .....  | 368        |
| Controllers .....                                   | 371        |
| alias_action .....                                  | 375        |
| Views .....   | 378        |
| Routes .....  | 385        |
| Custom Routes .....                                 | 386        |
| Filters .....                                       | 387        |
| Action Filters .....                                | 387        |
| Result Filters .....                                | 390        |
| Authorization Filters .....                         | 392        |

|  |            |
|--|------------|
| Exception Filters .....                                | 393        |
| Controller-wide Filters .....                          | 394        |
| Custom Action Filter Classes .....                     | 395        |
| Validations .....                                      | 396        |
| Inside the Model .....                                 | 396        |
| Inside the Controller .....                            | 396        |
| Inside the View .....                                  | 397        |
| Classic ASP.NET Features .....                         | 398        |
| A Word About Classic ASP.NET .....                     | 398        |
| <b>16 Silverlight</b> .....                            | <b>401</b> |
| Prepare Your Environment .....                         | 402        |
| Hello, Silverlight .....                               | 402        |
| The sl Tool: The Silverlight Application Creator ..... | 402        |
| The chr Tool: The Development Server .....             | 404        |
| Add Silverlight to a Web Page .....                    | 406        |
| XAML .....   | 409        |
| Layout .....   | 410        |
| Controls .....   | 411        |
| Adding Code .....                                      | 411        |
| Running XAML .....                                     | 411        |
| Retrieving Silverlight Elements .....                  | 412        |
| Event Handling .....                                   | 414        |
| Accessing the HTML Page and Window .....               | 414        |
| Graphics .....   | 415        |
| Media and Animations .....                             | 417        |
| Data Binding .....                                     | 419        |
| Static Data .....                                      | 419        |
| Dynamic Data .....                                     | 420        |
| Data Templates .....                                   | 422        |
| <b>17 Unit Testing</b> .....                           | <b>425</b> |
| The Tested Code .....                                  | 426        |
| Test::Unit .....                                       | 427        |
| Test Cases .....                                       | 427        |
| Assertions .....                                       | 428        |
| Setup and Teardown .....                               | 432        |
| Test Suites .....                                      | 433        |
| Running the Tests .....                                | 434        |
| RSpec .....  | 435        |
| Install RSpec .....                                    | 436        |
| Requiring Needed Libraries .....                       | 436        |

|   |            |
|---|------------|
| Running Tests .....                           | 437        |
| Creating a Behavior with describe .....       | 438        |
| Creating Examples with it .....               | 439        |
| Expectation Methods .....                     | 439        |
| Before and After .....                        | 442        |
| Cucumber .....                                | 443        |
| Installing Cucumber .....                     | 445        |
| Project Structure .....                       | 445        |
| Features .....                                | 446        |
| Scenarios .....                               | 447        |
| A Background .....                            | 452        |
| Tags .....                                    | 453        |
| Hooks .....                                   | 454        |
| A World .....                                 | 456        |
| Multilanguage .....                           | 456        |
| Executing Cucumber .....                      | 457        |
| <b>18 Using IronRuby from C#/VB.NET</b> ..... | <b>459</b> |
| Hello, IronRuby from CLR .....                | 459        |
| The Classes of the Process .....              | 461        |
| ScriptRuntime .....                           | 462        |
| ScriptEngine .....                            | 463        |
| ScriptScope .....                             | 465        |
| ScriptSource .....                            | 466        |
| Executing IronRuby from C#/VB.NET .....       | 468        |
| Executing an IronRuby File .....              | 468        |
| Executing IronRuby Code from a String .....   | 468        |
| Pass Variables to and from IronRuby .....     | 469        |
| Using IronRuby Objects .....                  | 470        |
| Using External Libraries .....                | 472        |
| <b>Part V Advanced IronRuby</b>               |            |
| <b>19 Extending IronRuby</b> .....            | <b>477</b> |
| Creating an Extension .....                   | 478        |
| Main Concepts .....                           | 478        |
| The Extension Project .....                   | 481        |
| Target Environments .....                     | 482        |
| Modules .....                                 | 482        |
| Classes .....                                 | 488        |
| Methods .....                                 | 491        |
| Constants .....                               | 501        |

|  |                |
|--|----------------|
| Using an Extension in IronRuby .....               | 501            |
| Building an IronRuby Extension.....                | 501            |
| Creating the Extension Visual Studio Project.....  | 502            |
| Adding Build Configurations .....                  | 502            |
| Creating the Actual Code .....                     | 504            |
| Creating the Ruby Programming Interface .....      | 506            |
| Generating the Library Initializer .....           | 508            |
| Using the IronRuby .NET Extension in IronRuby..... | 509            |
| <br><b>Index</b>                                   | <br><b>511</b> |



# About the Author

Shay Friedman works extensively with IronRuby and other dynamic languages, but got his start writing DOS scripts at age 8. He has been working professionally since age 16—first as a freelance Web developer, coding mainly in ColdFusion and ASP, and later as part of an information security team in an Israel Defense Forces (IDF) unit, where he developed in C#, C++, and Perl. Friedman moved on to join Advantech Technologies as a C# developer, and later as the leader of a Web development team at ActionBase. He currently works as a consultant and teacher of dynamic languages and ASP.NET at The Sela Group, conducting training courses around the world.

Feel free to contact the author through his website at <http://www.IronShay.com>.

# Dedication

*To Shira, you bring light to my life.  
You make me believe I can achieve anything.  
With you, I really can.*

*To my parents, Shimon and Hana, my sister, Tali,  
and my brothers, Ofer and Shraga, you taught me well  
and helped me become who I am today.*

# Acknowledgments

Writing a book is not an easy task at all. However, without the help of many people this task would have been much more difficult, on the edge of impossible. I want to thank these people for lending me a hand during the process of writing this book.

To Brook Farling, the nicest person you would ever meet and a great acquisitions editor as well, thanks for the support, the patience, and the understanding.

To Justin Etheredge, a .NET guru, thanks for the technical reviews, feedback, and answers.

To the IronRuby team—John Lam, Jimmy Schementi, Jim Deville, Shri Borde, Tomas Matousek, and Curt Hagenlocher—for helping me out with my hurdles while you were working hard to make IronRuby possible.

To the IronRuby mailing list participants, for assisting with every question I ran into.

To the team at Sams—Cindy Teeters, Mark Renfrow, Andy Beaster, and Keith Cline—for making this book possible.

To Shira, my better half, thanks for the unlimited support throughout this journey despite the odd working hours and stressful days. I would not have been able to do it without you.

# We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an executive editor for Sams Publishing, I welcome your comments. You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that I cannot help you with technical problems related to the topic of this book. We do have a User Services group, however, where I will forward specific technical questions related to the book.

When you write, please be sure to include this book's title and author as well as your name, email address, and phone number. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: [feedback@sampublishing.com](mailto:feedback@sampublishing.com)

Mail: Neil Rowe  
Executive Editor  
Sams Publishing  
800 East 96th Street  
Indianapolis, IN 46240 USA

## Reader Services

Visit our website and register this book at [informit.com/register](http://informit.com/register) for convenient access to any updates, downloads, or errata that might be available for this book.

# Introduction

The Ruby language was developed by Yukihiro Matsumoto, who began developing Ruby on February 24, 1993. His main reason for doing so was his dissatisfaction with the scripting languages at the time, languages such as Perl and Python. He designed Ruby to be intuitive, to be natural, and to follow the “principle of least surprise”—making developers enjoy writing code and focus on the creative part of programming instead of fighting the language to fit their needs.

Ruby 1.0 was released on December 25, 1996, exactly 1 year after the first public release (version 0.9.5) of Ruby. For the first year afterward, Ruby was mainly used inside Japan. Its use expanded outside of Japan a few years later, but it was still used by a small number of eager early adapters.

In 2006, David Heinemeier Hansson from 37signals released a web development framework named Ruby on Rails. This innovative MVC web framework made the difference. More and more developers started using Ruby on Rails to develop their web applications and in the process became familiar with the Ruby language, too. Following these newcomers, a phrase was coined to explain how most current Ruby developers have come to use it: I came for the Rails, but I stayed for the Ruby.

Since then, Ruby has become one of the most popular programming languages in the world and is being used by thousands of developers every day.

Ruby is a dynamic language. It combines ideas from Perl, Smalltalk, Eiffel, Ada, and Lisp to provide an intuitive, flexible, and simple-to-use language. Its strengths are in its permissive syntax and powerful built-in capabilities, especially metaprogramming capabilities. However, it never appealed to .NET developers because it lacked integration with .NET code. This situation has changed with the birth of IronRuby.

IronRuby is Microsoft’s implementation of the Ruby language. It runs on top of the Dynamic Language Runtime (DLR), which is a special dynamic language service provider that is built on top of the Common Language Runtime (CLR).

IronRuby provides seamless integration with .NET code. It enables you to use .NET objects in Ruby code just as if they were pure Ruby objects. This opens vast opportunities to the .NET world and to the Ruby world. Both sides gain the power and strength of the other and provide together a new and exciting development environment.

In this book, I take you through all aspects of IronRuby so that you learn how to best leverage the language for the simplest of tasks to the most advanced ones.

Part I, “Introduction to IronRuby,” is an overview of the different pieces that make IronRuby possible. You learn about where it all began and the main concepts of the Ruby language, the .NET Framework, and the DLR. Part I ends with a chapter that introduces you to IronRuby for the first time. In that chapter, you learn the basics about using IronRuby and witness the powerful capabilities it brings to your development environment.

Part II, “The Ruby Language,” is devoted to an in-depth tutorial of the Ruby language. The part starts with the basic syntax, goes on with Ruby object-oriented programming capabilities, and ends with advanced concepts and techniques.

In Part III, “IronRuby Fundamentals,” I add the Iron to Ruby. This part contains all the information you need about IronRuby .NET integration. I explain how every .NET item can be used from IronRuby—from variables to implementing .NET interfaces.

Part IV, “IronRuby and the .Net World,” is the practical part. It contains guides for how to use IronRuby in several different scenarios. Most of the current .NET and Ruby frameworks are explained, including WPF, ASP.Net MVC, Ruby on Rails, and Silverlight. In addition, other possible usages are explained, such as testing .NET code using Ruby’s different unit testing frameworks and running IronRuby code from .NET code.

The last part of the book, Part V, “Advanced IronRuby,” covers IronRuby advanced topics. If you want to extend IronRuby objects or to create .NET code libraries that fit better to Ruby code, you will be interested in what this part has to offer.

I believe that IronRuby can enhance your work and enable you to do things you have not done before. I hope you find this book helpful and informative and that you can exploit its contents in your own projects and development tasks.

# PART I

## Introduction to IronRuby

### IN THIS PART

|           |  |    |
|-----------|--|----|
| CHAPTER 1 | Introduction to the Ruby Language                  | 5  |
| CHAPTER 2 | Introduction to the .NET Framework                 | 13 |
| CHAPTER 3 | Introduction to the Dynamic Language Runtime (DLR) | 21 |
| CHAPTER 4 | Getting Started with IronRuby                      | 25 |

*This page intentionally left blank*

## CHAPTER 1

# Introduction to the Ruby Language

### IN THIS CHAPTER

- ▶ History of the Ruby Language
- ▶ Implementations
- ▶ Features

After a few years of using the same static programming language, we become apathetic to it. We think of its advantages as an obvious thing; and moreover, we ignore its disadvantages because we know there's nothing we can do about them.

We have to write all this code so that the single line of the code we really intended to write can be executed; we have to follow strict rules if we want our application to compile; and when our application gets bigger, it takes a few minutes to test any change in a single line of code.

When my interest in Ruby began to deepen, I had to remember to keep my mouth shut. The whole new world that was revealing to me was so different from what I was used to, and the built-in capabilities of the language just struck me.

In this chapter, I introduce you to the Ruby language. What you learn here about its capabilities will, I hope, pique your interest in the whole new world of opportunity that lies ahead of you with Ruby.

## History of the Ruby Language

Yukihiro Matsumoto, also known in the Ruby community as Matz, started the Ruby programming language in 1993. His goal was to make a language more powerful than Perl and more object oriented than Python. The language was designed for programmer productivity and fun. The result was eventually a mix of Perl, Smalltalk, Eiffel, Ada, and Lisp.



The name Ruby was preferred over Coral (the second possibility) because ruby the gemstone was the birthstone of one of Matsumoto's colleagues.

The first version, 0.95, was released in December 1995 and was quickly followed by three more versions in the next 2 days. One year later, in December 1996, Ruby 1.0 was released to the public.

Until 1999, Ruby was known mainly within the borders of its country of origin, Japan. After version 1.3 came out, interest began to increase outside of Japan, which led to the establishment of Ruby's first English mailing list.

It was not until 2006, when Ruby on Rails got the attention of the masses, that Ruby became widely known. Since then, Ruby has reached the top 10 of the popular languages list and continues to be used by thousands of developers worldwide.

The current stable Ruby version, 1.9.1, came out in January 2009.

## Implementations

The first and most popular implementation of the Ruby language is called MRI, which is short for Matz's Ruby Interpreter. MRI is written in C and runs on most operating systems, including Windows, Linux, and Mac. Recently, Microsoft Windows CE and Symbian OS were added to the supported operating systems, bringing Ruby to cellular phones as well.

MRI is an open source project and is totally free to use. You can even grab the code and contribute to the project. For more information about these subjects, take a look at <http://www.ruby-lang.org/en/community/ruby-core>.

Many other implementations have been introduced over the years: JRuby, which runs on top of the Java Virtual Machine; MacRuby; Rubinius; XRuby; and, of course, IronRuby.

Ruby has a test collection called *Ruby Spec* that describes the expected behavior of the language. When a new implementation is created, it is tested against the Ruby Spec to determine how close it complies to the expected language behavior.

## Features

Ruby is an object-oriented dynamic language. It offers several features as a result of that description, such as duck typing and an REPL (an interactive programming environment, explained later in this chapter). In addition, the Ruby language contains a few unique features, such as mixins and a huge collection of libraries. The next sections take you through the main features one by one.

### Dynamic Language

First of all, Ruby is a dynamic language. A dynamic language is a type of programming language. Like any other programming language, dynamic languages support statements, variables, and the handling of input and output. However, they differ from other programming languages in their special behaviors.

The “signature” behavior of dynamic languages is the significant preference of runtime over compilation time. They tend to execute behaviors, like code execution, during runtime rather than compilation time because there is no traditional compilation step.

This special approach creates opportunities and incorporates capabilities that are too complicated or even impossible to accomplish in other languages (for example, duck typing and an REPL). Another issue inherent in the discussion about the differences between static and dynamic languages is ceremony versus essence. Most static languages force you to write quite a bit of code before you ever get to the code you really want to write. For example, C# forces you to define a class and a method before you can start writing the “real” code. Dynamic languages are not like that. They enable you to write only the code you need; no class or method is required.

For example, if you want to write an application that writes “Hello World” to the screen, this is the code you use:

```
class Main
{
    static void main(string[] args)
    {
        Console.WriteLine("Hello World")
    }
}
```

In the Ruby language, writing the same application requires only a single line of code:

```
puts "Hello World"
```

Although it might be concluded from the terms, dynamic languages are not the opposite of static languages. They actually have more similarities than differences. Some even argue that some alleged static languages, such as C#, are actually dynamic languages because of their dynamic capabilities.

## Object Oriented

Ruby is a fully class-based object-oriented language. As such, it features classes, inheritance, encapsulation (using private, protected, and public directives), and polymorphism. Every value in the language is, at its root, an Object class instance, even numeric or Boolean values.

An especially interesting part of Ruby’s object-oriented capabilities is modules. A module can contain several classes and group them together with a single logical name, very similar to the concept of namespaces in C#. In addition, modules can contain methods. Modules with methods can be included in, or “mixed into,” every class and enrich these classes with their methods. This way you can declare and implement a certain behavior and spread it to multiple classes with a single line of code. This is the way Ruby adds enumeration capabilities to different built-in classes, for example.

For more information about Ruby's object-oriented capabilities, see Chapter 6, "Ruby's Code-Containing Structures."

## Duck Typing

Ruby uses an implicit type system. Therefore, you do not need to declare the type of your variables. However, this doesn't mean Ruby doesn't have types; Ruby has a dynamic typing mechanism. This means that the types will be calculated during runtime and will be enforced. This is the opposite approach to static typing, which is used commonly in static programming languages. Programming languages that use static typing perform type checks during compile time and prevent an application from compiling when a type error is found.

To understand Ruby's dynamic typing, take a look at the next code sample. It is okay because it doesn't fail any type policy validation:

```
my_var = 5
my_var = "IronRuby"
```

However, the next code raises an exception because when `my_var` is set with a string it becomes a `String` type and supports only `String` methods, which do not include the slash operator:

```
my_var = 5
my_var = my_var / 2 # OK
my_var = "IronRuby"
my_var = my_var / 2 # Error!
```

The implicit typing mechanism opens up the road for a feature called *duck typing*. The basic principle is this: If it sounds like a duck and swims like a duck, then it's probably a duck. Taking this sentence into the programming world, this means that every object should not be seen as an instance of class X. Instead, it should be seen as an object that responds to method Y.

For example, suppose we have a method that adds two objects together and returns the following result:

```
def add(a, b)
  result = a + b
  result
end
```

This method can obviously receive integers:

```
add(1, 3) # = 4
```

Moreover, it can receive floats, strings, or even a date and an integer combination:

```
add(1.5, 4.6) # = 6.1
add("Iron", "Ruby") # = "IronRuby"
add(Time.now, 60) # = adds 60 seconds to the current time
```

The main concept here is that it does not matter which type the parameters consist of. It just matters that they can be joined together with a plus operator.

Ruby offers a special method that can be used in such cases: `respond_to?`. This method returns true if the target object contains an implementation of a method with a given name. For example, the next sample code checks whether a string contains a definition of a plus operator:

```
"My string".respond_to?("+") # = true
```

## Metaprogramming

Ruby is a permissive language. It gives as much control as possible to the programmer, and programmers can generally reshape the language to their own needs instead of reshaping themselves to the language.

*Metaprogramming* refers to Ruby's capabilities in terms of modifying class implementation (built-in and custom classes), controlling code executing during runtime, and reflection functionality.

For example, Ruby can open a class and add, remove, or redefine its methods.

The next sample code adds a method to the built-in `Numeric` class, which prints the number in a friendly message:

```
class Numeric
  def print_friendly
    puts "The number is: #{self}"
  end
end
```

Now every number in the system responds to the `print_friendly` method—integers and floats as well as every other instance of a class that inherits from the `Numeric` class:

```
1.print_friendly # Prints "The number is: 1"
1.8.print_friendly # Prints "The number is: 1.8"
```

In addition to opening classes, Ruby has methods that are used as fallback methods. This means that whenever a method is called and it does not have an implementation, the call is redirected to the fallback method. The fallback method is called `method_missing`.

The obvious use for such is error handling: When an unimplemented method is called, we can log it or show some kind of a message to the user. This is a good use, but it doesn't take advantage of the great power that this mechanism holds. We can take it one step further and use `method_missing` to enable users to write method names that can be interpreted afterward.

For example, the next code makes it possible to call methods like `add_1_and_2` or `add_5_and_6` by using the `method_missing` fallback method:

```
class Sample
  def method_missing(name)
    # Convert the method name to string
    method_name = name.to_s
    # Run a regular expression on the method name
    # in order to retrieve the numbers
    result = /add_(\d)_and_(\d)$/.match method_name

    # If result is null, send the method to the base implementation
    # since the name of the method isn't valid for our operation
    super if result.nil?

    # Get the found numbers (result[0] contains the whole valid string)
    a = result[1].to_i
    b = result[2].to_i
    # Return the sum of the two numbers
    a + b
  end
end
```

## REPL

Ruby features a read-evaluate-print loop (REPL). REPL refers to the capability to receive commands and execute them instantly. Pretty much like the Windows command prompt, Ruby has an interactive console that allows writing Ruby code and executing it immediately. It also allows defining whole classes and using them.

The main console for IronRuby is accessible through the `ir.exe` file. Just run the file (which should appear on the IronRuby installation folder), and the IronRuby console loads.

Following is a sample session on the IronRuby console:

```
> ir
IronRuby 1.0.0.0 on .NET 2.0.50727.4927
Copyright (c) Microsoft Corporation. All rights reserved.

>>> 1+1
=> 2
```

```
>>> "IronRuby".reverse
=> "ybuRnorI"
>>> def add(a, b)
...   a + b
... end
=> nil
>>> add(1, 2)
=> 3
>>> exit
```

REPL is a great way to test code in a hurry or find out how a method reacts to a given input. In this book, you also see that you can add REPL capabilities to your applications (for example, WinForms, WPF, or Silverlight) and make testing and polishing them much easier.

## Available Libraries

Ruby has been out there for more than 13 years now. That's a lot of time for a programming language, and during this time (especially in the past 3 years), a lot of libraries were written for it. Most of them are free for use, and all you need to do to use them is to find them.

There are libraries that cover almost everything, and there is a good chance that any complex algorithm you are about to write has already been implemented and is just waiting for you to use it.

Several websites collect and list these libraries. The biggest are RAA (<http://raa.ruby-lang.org>), RubyForge (<http://rubyforge.com>), and GitHub (<http://github.com>).

For more information about Ruby libraries, see Chapter 7, "The Standard Library."

## Summary

The Ruby language is not a new language. It has been around for more than a decade and has proven itself to many as the right language for the job. It definitely has a wow factor when you start to discover its built-in capabilities.

Ruby offers many great features that help developers become more effective and happier with their programming language. This is why Ruby has made it into the top 10 programming languages in the world and continues to attract more and more converts.

*This page intentionally left blank*

## CHAPTER 2

# Introduction to the .NET Framework

Microsoft announcement about the .NET framework in 2002, came after long years that C++ had been the king of programming language for the Windows environment. Some people liked it, a lot of others didn't, to say the least. You had to do so much work to get the most common task done. The .NET Framework was released with a big promise and even a bigger concept: Leave the nasty stuff to the compiler and focus on your design and your targets.

The first version wasn't that great. However, more than 7 years have gone by, and we're already on version 3.5 of the .NET Framework, and eagerly awaiting 4.0. C# and VB.Net have already earned the trust of developers, and they both are now in the top 10 programming languages in the world.

The .NET Framework is the other half of IronRuby. It provides its architecture, services, and powerful frameworks, which enable the Ruby language to walk the miles it couldn't have walked before.

In this chapter, you learn about the .NET Framework, including where it came from and where it is now. You learn about its architecture and main concepts and its range of features.

## History of the .NET Framework

Just like spoken languages, programming languages evolve. That's a fact. When a language stops to grow and advance, it will slowly but surely lose all its fans and users. The .NET Framework, initially, represented the evolution of C++, VB,

### IN THIS CHAPTER

- ▶ History of the .NET Framework
- ▶ Overview
- ▶ Features



and ASP. In addition, the architecture of the .NET Framework offered an opportunity for other languages to be created on top of it.

C++, VB 6, and ASP weren't the only reasons for the .NET Framework, however. Microsoft doesn't live in a void, and the people at the headquarters couldn't miss Sun's new baby, Java. Java first came out in 1996, and was a breakthrough in terms of cross-platform deployment. Java's concept was "write once, run anywhere"—a concept that could not be achieved by any Microsoft languages at the time.

At the core of Java is the Java Virtual Machine, known as the JVM. The code you write goes first to the JVM, where it is translated to machine code and executed. Thus, the only thing that changes when moving between Windows and Linux, for example, is the JVM implementation, and not your code.

Microsoft joined the choir and wrote its own JVM implementation, MSJVM. Moreover, Microsoft created its own version of Java and named it J++. It even had an IDE, which was called Visual J++.

Sun claimed that Microsoft did not conform to the Java standards and had added Windows-only extensions to their MSJVM. As a result, Sun initiated litigation against Microsoft for failing to adhere to Java's license agreement. After a long battle, Microsoft and Sun signed an agreement that practically meant that Microsoft's Java implementation would disappear in a few years. Since then, Microsoft has stopped developing its JVM implementation, and J++ has been discontinued. The last Java-related action from Microsoft came with J#, which was intended to help J++ developers to move on to the new .NET Framework. However, the language is now discontinued, and its last version was released with Visual Studio 2005.

After the Java experience, Microsoft started to work on its own idea: the .NET Framework. Initially, it was called Next Generation Windows Services (NGWS), but that name was later changed to the .NET Framework. Microsoft wanted industry approval for their new project and sought out industry partners. As a result, Intel and HP joined Microsoft and cosponsored the implementation of C# and the Common Language Infrastructure (CLI). Moreover, C# and CLI specifications are ratified by ECMA.

Unlike Java, Microsoft doesn't share its CLI code and does not encourage the creation of other runtime environments. Others, however, such as the Mono project (an open source project led and sponsored by Novell), intend to run .NET code on various platforms, including Linux, BSD, UNIX, Mac OS X, and Solaris.

The first version of the .NET Framework came out in 2002, followed by five more versions in the next 7 years. By the time Microsoft released the .NET Framework, along with C#, VB.Net, and ASP.Net, it was clear to all that programming in the Microsoft world would never be the same again.

## Overview

The .NET Framework was created based on the experience, bad experience, of developers with the language prior to it. For example, C and C++ weren't cross platform. You had to compile the code on the target operating system and even modify it for the application to work on the specific system.

The .NET Framework brought a whole new architecture that resolves the problem completely. The .NET architecture separates the framework into three different pieces: the language, such as C# or VB.Net; the class library; and the Common Language Runtime (CLR).

Figure 2.1 presents the architecture of the .NET Framework.

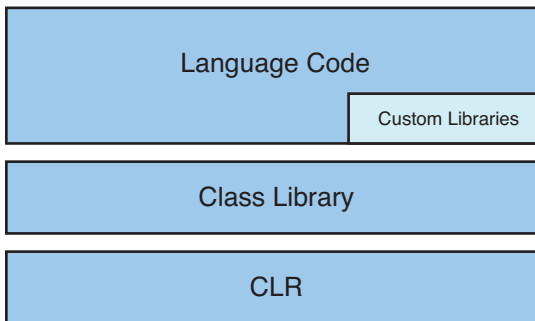


FIGURE 2.1 The architecture of the .NET Framework.

The separation is also hierarchical. The language code is on top and uses the base class library. Then all that code goes through the CLR and gets executed.

Figure 2.2 shows the flow of .NET code until it is executed on the target machine.

The flow of program execution starts with the language code. This could be any .NET language that uses the CLR. .NET dynamic languages, unlike static languages that communicate directly with the CLR, go through another layer before the CLR (the DLR, which is discussed in the next chapter).

Next we compile the code via Visual Studio or `csc.exe`. Our .NET code is compiled into a DLL or EXE file (according to the project type). This file is called a *.NET assembly*. The assembly doesn't hold the actual code we have written; it contains a Common Intermediate Language (CIL, also known as IL or MSIL) code. CIL code is part of the Common Language Infrastructure (CLI), and it is the language the CLR is familiar with. For more information about CIL, see the "Features" section later in this chapter.

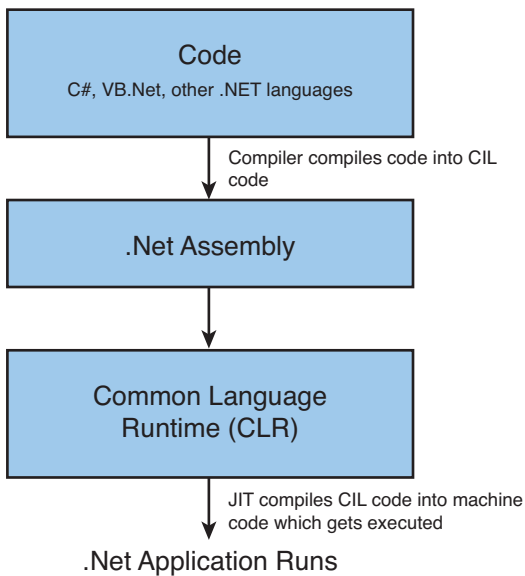


FIGURE 2.2 The road of .NET code until it is executed on the target machine.

After the code is compiled and an assembly is created, we can run it. When the assembly is run, it goes through the CLR, which is not a part of the application code or assembly and comes in a separate installation package. A feature called *JIT compiling* (just-in-time), which is a part of the CLR, converts the IL code to CPU instructions and executes them.

From then on, while the application is running, the CLR controls the code behind the scenes and provides several services to it (for example, memory management, security, and caching).

## Features

Many .NET Framework features were designed to make the development process faster and more efficient. The framework was also built so that applications could be run locally, remotely, and distributed via the Internet. In addition, you notice that some of the features are fixes for major problems in development prior to the .NET Framework.

### Common Language Infrastructure

The Common Language Infrastructure (CLI) is a specification developed by Microsoft that contains the details of the different parts of the CLI:

- ▶ **Common Type System (CTS):** Describes types and operators that can be used by all CTS-compliant languages written on top of it.
- ▶ **Metadata:** Holds application structure information and references to CTS types, which enables cross-language communication.

- ▶ **Common Language Specification (CLS):** Describes a set of rules that any CLS-compliant language must conform to.
- ▶ **Virtual Execution System (VES):** Responsible for loading and running programs written for the CLI and does so by using the application metadata.

The CLI has several different implementations. The most noted one is the Common Language Runtime (CLR), which is Microsoft's commercial implementation of the CLI. It is distributed for free and works on Windows operating systems.

Mono, an open source project led by Novell, is another implementation. Its target is to make it possible to run .NET applications on non-Windows platforms (including Linux, Mac, BSD, and more).

Microsoft has some other CLI implementations, too; for example, the .NET Compact Framework, which intends to bring .NET applications to mobile devices (and Xbox 360), and Silverlight, which implements a part of the .NET Framework for use inside web browsers.

### Common Language Runtime

The CLR is Microsoft's commercial implementation of the CLI specs that runs on Windows operating systems.

When .NET language code is compiled, the compiler converts the code to CIL, also known as IL or MSIL code. This way, all code that reaches the CLR is CIL code. During runtime, this CIL code is converted to machine code and gets executed. The component responsible for compiling CIL into native machine code is called the JIT compiler.

### Common Intermediate Language

CIL, formerly called Microsoft Intermediate Language (MSIL), is a low-level programming language that every .NET language compiles to.

CIL is an object-oriented assembly language and is entirely platform-independent. Therefore, the CLI can compile CIL code on different platforms.

To get the feeling of what CIL looks like, look at the next C# code. This code uses the base library `System.Console` class and prints "Hi from C#" on the console:

```
private void PrintHello()
{
    Console.WriteLine("Hi from C#");
}
```

The next code is the equivalent code in CIL:

```
.method private hidebysig instance void PrintHello() cil managed
{
    .maxstack 8
    L_0000: nop
    L_0001: ldstr "Hi from C#"
}
```

```

L_0006: call void [mscorlib]System.Console::WriteLine(string)
L_000b: nop
L_000c: ret
}

```

## Assemblies

A .NET assembly is a file (usually with an .exe or .dll extension) that contains .NET code written in CIL.

A single assembly can contain multiple code files, and each one of them is called a *module*. Code modules can, in principle, be written in different .NET languages and create an assembly together, but currently Visual Studio does not support that.

This possibility is a direct result of the .NET framework architecture that compiles code from different languages to the same CIL.

Every assembly has a name that consists of four parts:

- ▶ **Short name:** The name of the file without its extension.
- ▶ **Culture:** The culture of the assembly. Generally, assemblies are culture neutral, but some assemblies do contain language-specific strings and resources that do have a culture.
- ▶ **Version:** Every assembly has a version built from four values (major, minor, build, and revision). These values are separated by dots, and so a version value will look like 1.5.67.13422.
- ▶ **Public key token:** This is used when an assembly is signed. It contains a 64-bit hash of the public key, which corresponds to the private key used when signing the assembly. Signed assemblies are said to have a strong name. Signing the assembly makes it more secure against possible spoofing attacks (in which the assembly is maliciously replaced).

### The Global Assembly Cache

In C++, all developers learned in their first week of training what DLL hell was. The lack of standards and the lack of a central place for libraries made it really difficult to confirm that all of them existed and whether they were in the correct version. This is exactly what the Global Assembly Cache (GAC) addresses.

The GAC is a central place for all shared .NET libraries on a single machine. Assemblies that are put in the GAC must have a version and be strong named.

It is possible for various versions of the same assembly to exist in the GAC. The application that uses it should refer to a specific version if needed; otherwise, it will be supplied with the latest one.

## Base Class Library

The Base Class Library (BCL) is the standard library of the .NET framework available to all languages written on top of it or using it.

The BCL contains various different libraries for many different needs. These include libraries that provide basic needs such as basic types (`String`, `Boolean`, and so on) or collection types (`List`, `Dictionary`, `Stack`, and so forth). More advanced libraries are available, too, including file handling, network protocols communication, database interaction, and many more.

All the BCL library names start with *System*. For example, the basic library is called `System`, and the library that contains collections is called `System.Collections`.

## Security

The .NET framework provides two methods of security: Code Access Security (CAS), and validation and verification.

CAS uses an object called *evidence* to find out which permissions are granted to the code. An evidence can be the directory where the assembly is located, the URL from where the assembly was launched, a strong name, or other available evidence types. (Custom evidence implementation can also be used.) When code runs, permission demands are made to determine whether the code really has the requested permissions. Permissions are set in permission sets and can control almost anything (for example, blocking file handling or using network protocols).

Another security mechanism is validation and verification. When an assembly is loaded, the CLR validates that the assembly contains valid metadata, CIL, and internal tables. In addition, the CLR tries to verify that the assembly code doesn't do anything malicious.

## Memory Management

One of the biggest performance and security problems in C++ was faulty memory management. The .NET Framework takes care of that and removes that headache for the developer.

The CLR contains a component called *Garbage Collector* (GC). This component wakes up when there is a need to free up some memory, determines the object to release, releases it (them), and goes back to sleep.

The decision of which objects to release is made according to their references in the code. If an object is referenced directly or via a graph of objects, it is considered in use. Otherwise, it is candidate for release.

The candidates are released according to their generation. A generation of an object is actually the number of times it has “survived” a garbage collection and starts from zero. The candidates with the lowest generation are the ones that will be released first. The assumption is that older objects that have survived more garbage collections are more essential to the application than newly created ones.

## Frameworks

The .NET Framework incorporates numerous frameworks that Microsoft has released over the past few years. Each of these frameworks centralizes and standardizes the development of a single field:

- ▶ **Windows Forms:** Provides access to Microsoft Windows graphical elements
- ▶ **ASP.Net:** A web development platform.
- ▶ **ADO.Net:** A data framework that provides base classes to connect and use data taken from a data source.
- ▶ **Windows Presentation Foundation (WPF):** An enhanced graphical framework to create GUIs. WPF graphics are based on DirectX.
- ▶ **Windows Communication Foundation (WCF):** Provides classes to simplify building connected applications.
- ▶ **Windows Workflow Foundation (WF):** A framework that provides classes to define, execute, and manage workflows.
- ▶ **Windows CardSpace:** A framework that provides a unified identity mechanism to .NET applications.

These frameworks and others make the process of writing applications with the .Net framework easier and faster, leaving more time to invest in designing and proofing the application.

## Summary

The .NET Framework was a great leap ahead for Microsoft and for its developer community. It provided solutions that had never been seen before and led the entire programming world forward.

Microsoft continues to improve and enhance the .NET Framework, with new versions coming every year and a half on average. Within a few months of this writing, the next version of the .NET Framework (4.0) will most likely be released (and might have already been released by the time you read this). It will contain several improvements and exciting new features.

In this chapter, you were introduced to the .NET Framework. You learned about its history, its workflow, and its main features. The next chapter introduces you to the Dynamic Language Runtime, which is the .NET Framework component that enables dynamic languages to run on top of the .NET Framework.

## CHAPTER 3

# Introduction to the Dynamic Language Runtime

When people hear about IronRuby for the first time, they express various different expressions.

Some think it is great, some think it is nice, and some just don't have a determined opinion. (After all, most programmers prefer not to think too much about how something is actually implemented.) This chapter is about to pay respect to the dark horse of dynamic languages in the .NET Framework: the Dynamic Language Runtime (DLR).

As a result of the rise in interest and popularity of dynamic languages in the past few years, the .NET Framework was left behind, supporting only static languages. Developers who wanted to use the .NET Framework from their favorite dynamic language found it hard to do so.

The DLR came to solve this exact issue. It is the component that enables dynamic languages to be implemented on top of the Common Language Infrastructure (CLI). Thus, it enables interoperability between static .NET languages and dynamic .NET languages (and between different dynamic .NET languages, too).

It is designed to serve dynamic languages and provide them with the tools they need to become members of the .NET Framework.

In this chapter, you learn about the DLR's architecture various features.

### IN THIS CHAPTER

- Overview
- Features



## Overview

The Dynamic Language Runtime (DLR) was firstly introduced in 2007 at the MIX conference. It is still in development (version 0.91 in October 2009) and is planned to reach the version 1.0 milestone in the next few months.

The DLR is a set of libraries whose goal is to support dynamic language implementations on the .NET Framework. Its main goal is to make it possible for dynamic language developers to use .NET in their dynamic language of choice and vice versa (.NET static language developers who want to add dynamic capabilities to their platform).

Before the DLR was announced, several attempts were made to port dynamic languages to the .NET Framework (for example, Ruby.Net, IronPython 1.0, S#, and Phalanger). However, most of them didn't reach a maturity level because of the complex work to provide dynamic capabilities on top of the static-language-targeted Common Language Runtime (CLR). The DLR standardizes the implementation of dynamic languages on the .NET Framework and provides documentation and samples that make the process of implementing languages on top of it much easier than it was before.

Another important fact about the DLR is that unlike most other Microsoft products, it is an open source object. The sources are available on CodePlex and are published under the Microsoft Public License (MS-PL). You can get the sources at <http://www.codeplex.com/dlr>.

The DLR is composed of three big conceptual parts: a common hosting model, runtime components, and base classes for language implementations.

Figure 3.1 shows the DLR architecture.

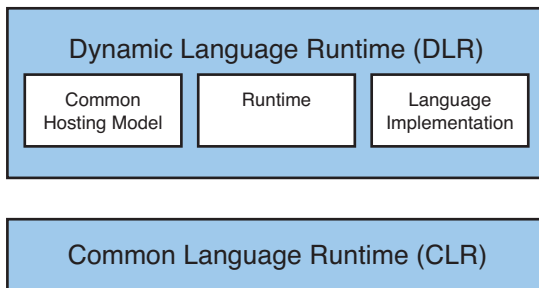


FIGURE 3.1 The Dynamic Language Runtime architecture.

The common hosting model provides developers a way to host the DLR and the languages written on top of it inside their applications. It is also used to host the DLR on different hosts, such as Silverlight.

The runtime components provide classes and utilities for the execution of dynamic language code. They contain, for example, caching mechanisms that make the code run faster.

The last part is the language implementation aspect of the DLR. This is where the heart of the DLR framework is located: shared ASTs (expression trees). It also contains the base classes for dynamic language implementations, like the language compiler that converts the dynamic language code to expression trees.

## Features

The DLR main features were introduced in the “Overview” section, earlier in this chapter. In this section, I explain them and their internal components in more detail.

### Common Hosting Model

The common hosting model contains several classes that make it possible to host the DLR in applications. Therefore, you can use the DLR hosting model to execute dynamic language code from your static-language-driven application.

The common hosting model consists of the following classes:

- ▶ **ScriptRuntime:** The starting point of the hosting model. Contains the global object of the application, the runtime configuration, and the referenced assemblies. The script runtime also provides methods for getting the specific-language engine.
- ▶ **ScriptEngine:** The hard worker of the hosting model. It represents a dynamic language, and every interaction with the language goes through the engine instance.
- ▶ **ScriptScope:** Represents a context. These hold variables and enable to manage them. The scope can be used with an engine to give a context to the running code. It also helps in isolating the context because running code with one script scope instance does not affect other script scope instances.
- ▶ **ScriptSource:** Represents dynamic language source code. Offers methods to read and execute the source code.

For more information about the classes of the common hosting model and their usage, see Chapter 18, “Using IronRuby from C#/VB.Net.”

### Runtime Components

The runtime components are responsible to execute dynamic language code. In addition, they are responsible for making the code run faster. The runtime components are as follows:

- ▶ **Call sites:** Call sites increase performance by caching operations and calls in dynamic code. The call site stores a rule for each call based on the argument characteristics. This way, if you send two `fixnums` to a method 10 times, the call will be interpreted only once and will be immediately returned in the next 9 times.

- **Binders:** When a dynamic call needs to be interpreted, it is sent to the binder. The binder converts the code to an expression tree and returns it to the call site for caching and further work.
- **Rules:** Represents a single cache object in a call site. Contains information about how to invoke a single operation.

## Language Implementation

The language implementation part includes the base classes for dynamic language implementations. This part is the main part in terms of a specific dynamic language. In case you are interested in writing your own language on top of the DLR, this part is what you should care about most.

The language implementation part consists of the following:

- **Shares ASTs (expression trees):** This is a central piece in the DLR. The DLR team took LINQ's expression trees, enhanced and upgraded them, and made them fit for presenting dynamic language calls.  
  
Actually, the code you write in IronRuby (or any other DLR dynamic language) is converted to an expression tree before it is executed. This way, the DLR compiler doesn't need to be familiar with any dynamic language, only with the expression trees.
- **Language context:** The actual language implementation. The language context contains the code that converts the dynamic language code to an expression tree that can be compiled and invoked afterward.
- **Compiler:** Compiles DLR expression trees and returns a delegate for invoking them.

## Summary

In this chapter, you learned about the DLR and its components. When developing with IronRuby, you will not run into the DLR often because it is the background worker of dynamic languages in the .NET Framework, just like the CLR for static .NET languages.

However, just like people research their family history to better understand themselves, it is good to know the roots of your programming language to understand it on a deeper level.

In conclusion, the DLR took the .NET Framework to the next level. It opened the .NET world to developers who hadn't had any connection to it before and thus created numerous new opportunities for interoperability and cooperation.

Now that you know all the background you need (Ruby, the .NET Framework, and the DLR), all that is left to learn is the main subject of this book: IronRuby.

## CHAPTER 4

# Getting Started with IronRuby

**I**ronRuby is Microsoft's implementation of the Ruby language on top of the DLR. Its main goal is to provide seamless interoperability between Ruby and the .NET Framework.

IronRuby combines the powers of both the .NET Framework and the Ruby language. On the one hand, it contains the built-in capabilities of Ruby, and on the other hand, it is capable of using the wide variety of frameworks and libraries of the .NET Framework. The combination opens a whole new set of opportunities to both Ruby and .NET developers.

In this chapter, you get your first taste of IronRuby. You install it, read an overview of the language and its tools, and start to discover the power it brings to the .NET family.

## Overview

IronRuby is Microsoft's implementation of the Ruby programming language. It is built on top of the DLR and provides seamless integration between Ruby code and .NET Framework code. It is compatible with Ruby 1.8.6 and runs on .NET Framework 2.0 Service Pack 1 and above.

IronRuby was first announced on April 30, 2007, at the MIX conference. *Iron*, in its name, is actually an acronym and stands for "implementation running on .NET."

IronRuby is supported by the Common Language Runtime (CLR) and Mono, which means that it can be run on Windows, Linux, UNIX, BSD, Mac, and all other operating systems that are supported by Mono. Apart from operating

## IN THIS CHAPTER

- Overview
- Installing IronRuby
- Executables and Tools
- Development Environments
- The Power of IronRuby

systems, IronRuby can also be run from the browser using Silverlight. (See Chapter 16, “Silverlight,” for more about IronRuby and Silverlight.)

IronRuby is an open source project and is released with full source code under the Microsoft Public License (MS-PL). The code is hosted on GitHub and can be downloaded from the CodePlex site, too. Because it is an open source project, the IronRuby team is looking for contributions both in bug fixing and library implementation. Look at the contribution page on IronRuby's GitHub home page to see how you can help (<http://wiki.github.com/ironruby/ironruby/contributing>).

## Installing IronRuby

IronRuby runs on .NET Framework 2.0 Service Pack 1 and above. Therefore, you need to have .NET Framework 2.0 SP1 or above installed on your machine before you start. Same goes for every machine on which you deploy your IronRuby applications.

You can download .NET Framework 2.0 SP1 from <http://www.microsoft.com/downloads/details.aspx?familyid=79BC3B77-E02C-4AD3-AACF-A7633F706BA5>.

When you have the correct framework, we can move on and install IronRuby. The recommended method of installing IronRuby is by using the IronRuby installer. Follow the next steps to do so:

1. Visit <http://www.ironruby.net/download> and click the Download IronRuby link.
2. The downloaded file is an MSI file. Double-click it to start the installation.

The first input you will be asked to enter is the installation folder for IronRuby. The default is your program files directory\IronRuby. The folder you choose will be referenced as the IronRuby installation folder throughout this book.

During the installation, you will be asked to select the features to install. The features that will be presented are as follows:

- ▶ **Runtime:** The main files of IronRuby. This feature is required.
  - ▶ **Standard Library:** Ruby standard libraries. Needed if the standard libraries are used in IronRuby code.
  - ▶ **Samples:** Sample IronRuby applications like WPF and PowerShell samples.
  - ▶ **Silverlight Binaries:** Binaries needed for using IronRuby in Silverlight applications.
  - ▶ **Add IronRuby to %PATH%:** Adds IronRuby binaries path to the PATH environment variable. This spares the need to provide full path to IronRuby executables when they are called from the command line.
3. After approving all steps, the installation of IronRuby takes place.

Another option to install IronRuby is manually. IronRuby can be downloaded as a zip package. This lets you fully control the installation process but also forces you to execute the automatic tasks manually (optionally).

Follow the next steps to install IronRuby manually:

1. Visit IronRuby's CodePlex homepage at <http://ironruby.codeplex.com> and click on the Downloads button on the page menu. In the downloads page, choose to download the IronRuby ZIP package.
2. After the download is complete, extract this Zip file to the folder in which you want to place IronRuby (for example, C:\IronRuby). This folder will be referenced as the IronRuby installation folder throughout this book.

You're actually done now and can start using IronRuby. However, if you want IronRuby to be available from every location on your Windows system, you want to add the <installation folder>\bin path to the Windows PATH environment variable. Be aware, however, that this can be done only if you have administrative privileges.

To do that, follow the next steps:

1. Navigate to Start > My Computer and right-click My Computer. On the menu, choose Properties as presented in Figure 4.1.

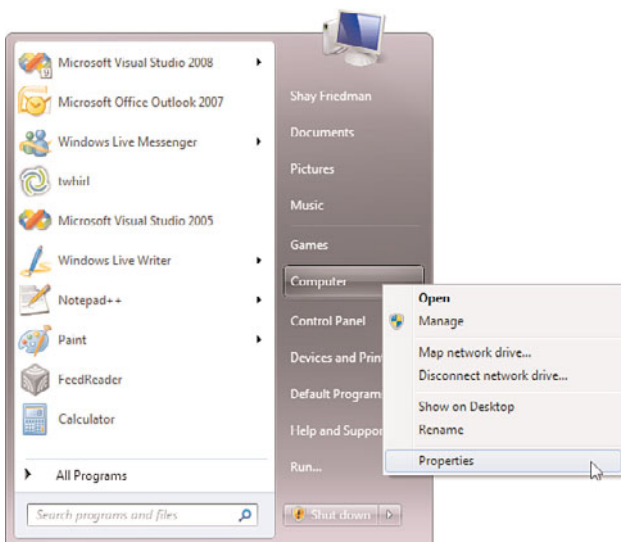


FIGURE 4.1 My Computer Properties Menu Item.

2. In the open dialog, click Advanced System Settings as presented in Figure 4.2.



FIGURE 4.2 The Advanced System Settings Link.

3. Click the Environment Variables button as presented in Figure 4.3.

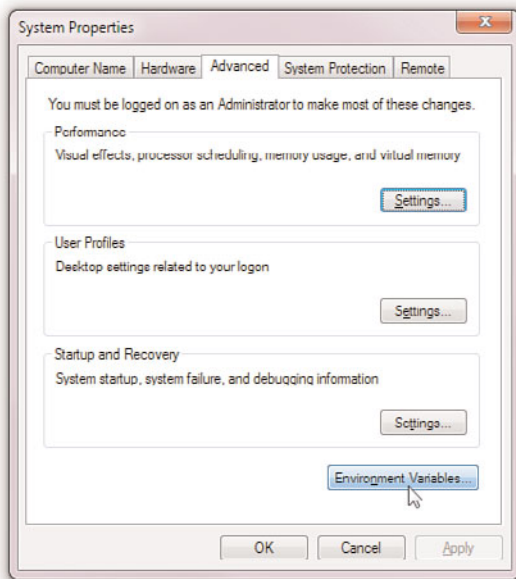


FIGURE 4.3 The Environment Variables button.

4. Find Path in the System Variables section (the lower part), select it, and click Edit as presented in Figure 4.4.
5. In the Edit System Variable dialog, place the cursor at the end of the Variable Value field and add a semicolon (;) and **<IronRuby installation folder>\Bin**. For example, if you've extracted IronRuby to C:\IronRuby, you add ;C:\IronRuby\Bin to the Variable Value field as presented in Figure 4.5.

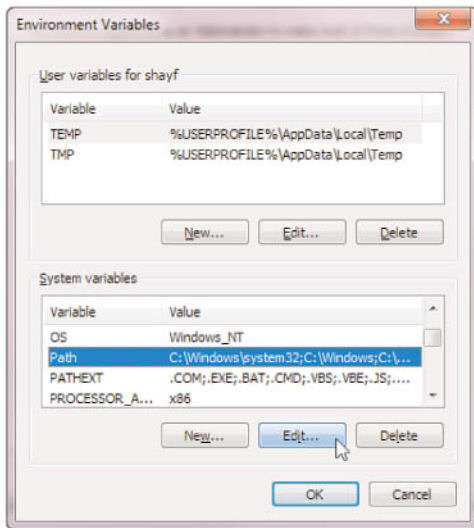


FIGURE 4.4 The PATH Environment Variable.

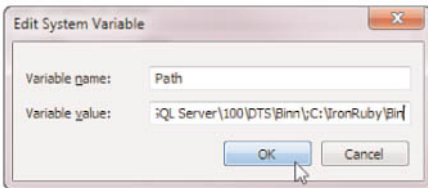


FIGURE 4.5 Setting the PATH Environment Variable.

6. Click OK on all the dialogs you've opened during the process to save the new setting.

Congratulations, IronRuby is now installed on your machine.

## IronRuby Folders

After you extract IronRuby to the desired installation folder, you notice several folders there. Table 4.1 lists and describes the folders.

## Getting the Sources

If you'd like to go deep into IronRuby and go through its code, you need to download the source code of IronRuby. The source code is hosted on GitHub and uses Git as its source control application. You can download the source code in a Zip format from <http://github.com/ironruby/ironruby/zipball/master>.



TABLE 4.1 IronRuby Folders and Their Roles

| Folder      | Description   |
|-------------|---|
| Root        | Contains license files, release notes, and the package readme file                                |
| Bin         | Contains the IronRuby binaries, executables, and tools  |
| Libs        | Contains the standard libraries, including special IronRuby libraries and the RubyGems repository |
| Samples     | Contains a few IronRuby samples and the IronRuby tutorial application                             |
| Silverlight | Contains binaries, samples, and tools for embedding IronRuby in Silverlight                       |

To download the sources, you do not have to install Git on your machine. You need to do so only to contribute to the IronRuby code. For more information about how to contribute and how to use Git, look at the IronRuby wiki at <http://wiki.github.com/ironruby/ironruby>.

## Executables and Tools

IronRuby comes with several different executables and tools. All of them are located under the Bin folder in the IronRuby installation directory.

Table 4.2 lists and describes the executables and tools in the IronRuby Bin folder, and then the following subsections take a closer look at them.

TABLE 4.2 IronRuby Executables and Tools

| File                      | Description   |
|---------------------------|---|
| ir.exe<br>and<br>ir64.exe | The main IronRuby executable. It is the IronRuby interpreter file. ir64.exe is for 64-bit systems.                                    |
| iirb.bat                  | The IronRuby REPL (read-evaluate-print loop) console.   |
| igem.bat                  | Used to work install and manage RubyGems. See Chapter 8, “Advanced Ruby.” for more information about RubyGems.                        |
| irackup.bat               | Runs Rack, which is a Ruby framework that simplifies the way of interacting with different Ruby web servers.                          |
| irake.bat                 | Executes the Rake. See Chapter 8 for more about Rake.   |
| irails.bat                | Used to create a Ruby on Rails application. See Chapter 14, “Ruby On Rails.” for more about Ruby on Rails.                            |
| irdoc.bat                 | Runs RDoc, which is a Ruby tool to create formatted documentation out of Ruby code. The output can be plain text or a formatted HTML. |
| iir.bat                   | Ruby tool to read the textual documentation of Ruby objects. (The documentation is created by RDoc.)                                  |

## The IronRuby Interpreter (ir.exe)

The IronRuby interpreter is the heart of IronRuby. Everything goes through it. For example, all the tools mentioned in Table 4.2 eventually run `ir.exe`.

The IronRuby interpreter can run Ruby files as well as a REPL console. These are two different modes, and so I discuss each of them separately.

### REPL Console Mode

The REPL console mode opens a console where you can write code and execute it immediately.

To run IronRuby in the REPL console mode, follow these steps:

1. Click Start > Run.
2. Type `cmd` and click OK. The command prompt opens.
3. If you haven't added the IronRuby installation folder to the Path system variable, navigate to `<installation folder>\Bin`. If you have updated the Path system variable, skip this step.
4. Type `ir` and press Enter.

The IronRuby REPL console opens. You can now write Ruby code there (for example, `puts 'hello world'`) or even write whole classes and use them.

The format is simple. Each line where you can write a Ruby statement starts with a triple right-angle sign (`>>>`). If the line is not assumed to be continued (like method or class definitions), when you press Enter its output (if any) is printed to the console and the next line contains the return value of the statement preceded by an equal or greater than sign (`=>`). If the line is assumed to be continued, the next statement line starts with an ellipsis (`...`).

### THE RETURN VALUE OF METHODS THAT DO NOT RETURN ONE

You notice that after executing methods with no return value, such as `puts`, the console still shows a return value: `nil`. This is really the return value.

Every method in Ruby returns a value. If the method code doesn't return a value, the method returns `nil` to the caller.

Figure 4.6 shows an REPL console session.

The REPL console mode has some specific command-line switches that can be used when running `ir.exe`. Table 4.3 lists and describes the switches. Be aware that the switches are case sensitive.

These are the REPL mode-only switches. All other switches appear on Table 4.4, and some can be used in this mode, too.

```

Administrator: C:\Windows\system32\cmd.exe - ir

C:\>ir
IronRuby 0.9.1.0 on .NET 2.0.50727.4927
Copyright (c) Microsoft Corporation. All rights reserved.

>>> puts "Hello World"
Hello World
=> nil
>>> str = "Hi"
=> "Hi"
>>> puts str
Hi
=> nil
>>> def sample_method(a, b)
...   return a + b
... end
=> nil
>>> sample_method(1,3)
=> 4
>>> 
=> ■

```

FIGURE 4.6 A screenshot of an IronRuby REPL console session.

TABLE 4.3 ir.exe REPL Mode Command-Line Switches

| Switch            | Description   |
|-------------------|---|
| -                 | Makes the REPL console colorful.  |
| X:ColorfulConsole | Prompt signs such as >>> and ... appear gray, errors red, and warnings yellow; messages (like the banner on top) appear cyan, and all other output uses the default console output color. |

## File Execution Mode

This mode executes a given file. Its format is as follows:

```
ir [options] [file path] [file arguments]
```

All switches and ir.exe command-line arguments should be placed before the file path. Otherwise, they will be considered arguments of the file and will be passed to it rather than to the interpreter.

The simplest way to execute an IronRuby file is by passing only its path to the interpreter. For example, the next command executes the file test.rb, which is located within the current directory:

```
ir test.rb
```

Along with this way, IronRuby provides several command-line arguments that affect the way the code executes. Table 4.4 lists and describes available arguments for the `ir.exe` file execution mode.

TABLE 4.4 `ir.exe` Command-Line Arguments for File Execution Mode

| Argument                      | Description   |
|-------------------------------|---|
| -d<br>or<br>-D                | Debug mode. Allows using breakpoints in Ruby code with the Visual Studio debugger.  |
| -e "command"                  | Executes the command and exits. Several -e arguments are allowed. When used, the file path should not be passed and will be ignored if it exists.<br>For example:<br><code>ir -e "str = 'Hello World'" -e "puts str"</code> |
| -I "directory"                | Includes the given directory in the <code>\$LOAD_PATH</code> variable. This means that it will be included in the search paths for required libraries.  |
| -r "library"                  | Requires the library before executing the file.<br>For example:<br><code>ir -r "csv" test.rb</code>   |
| -v                            | Prints the IronRuby version on the first line.  |
| -w                            | Turns on warnings in the default level (verbose).   |
| -W[level]                     | Sets the warning level. 0 = silence, 1 = medium, 2 = verbose (default).<br>For example:<br><code>ir -W1 test.rb</code>  |
| -K[kcode]                     | Specifies KANJI (Japanese) code set. E or e = EUC, S or s = SJIS, U or u = UTF8.<br>For example:<br><code>ir -KU test.rb</code>   |
| -trace                        | Enables Ruby tracing capabilities.  |
| -profile                      | Enables profiling. When this switch exists, a <code>profile.log</code> file will be added to the directory of the executed file with profiling information about the latest execution.                                      |
| -18<br>or<br>-19<br>Or<br>-20 | Run IronRuby in Ruby 1.8 compatibility mode (default), Ruby 1.9, or Ruby 2.0 compatibility mode accordingly.<br>Ruby 2.0 doesn't currently exist, so this switch is for future release only.                                |

TABLE 4.4 ir.exe Command-Line Arguments for File Execution Mode

| Argument                 | Description  |
|--------------------------|--|
| -X:ExceptionDetail       | In this mode, every exception is presented with a full call stack.   |
| -X:NoAdaptiveCompilation | Disables adaptive compilation feature. This affects performance (for the worse).   |
| -X:PassExceptions        | In this mode, exceptions are not caught by the interpreter. This means that in case of an exception with no handling in the script, the application crashes.   |
| -X:PrivateBinding        | Enables binding to private members of CLR objects. Chapter 9, “.Net Interoperability Fundamentals,” discusses the uses of this switch.   |
| -X:ShowClrExceptions     | When this mode is on, a CLR exception part is added to every exception with full exception details (the exception.ToString output).  |
| -X:CompilationThreshold  | Specifies the number of iterations before the interpreter starts compiling the code.<br>Should be followed by a number.<br>Note that this switch can affect performance dramatically. Hence it is not recommended to use it when not needed. |
| -h                       | Shows all available command-line arguments with a short description. When this exists, the file or REPL console does not run.  |

## Development Environments

Support for IronRuby in Visual Studio is not available in IronRuby 1.0. Such support is not in Microsoft’s current plans, and no one can really promise it will be in the near future.

However, the Ruby language already has several IDEs available. This section discusses some of them so that you can choose the one that best fits your needs.

### Ruby in Steel

This commercial add-on to Visual Studio by SapphireSteel makes developing Ruby applications inside Visual Studio much more natural. It adds new Ruby project types, intellisense, code snippets, and syntax highlighting.

Figure 4.7 is a screenshot from Visual Studio that shows the syntax highlighting and intellisense capabilities of Ruby in Steel.

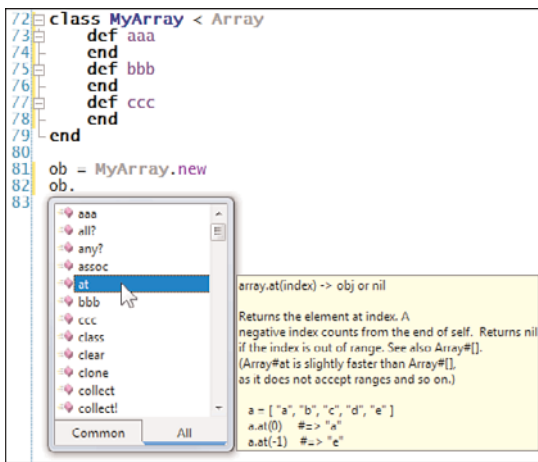


FIGURE 4.7 Ruby in Steel syntax highlighting and intellisense.

Although you cannot run IronRuby with Ruby in Steel out of the box, it is possible to alter the solution settings to execute `ir.exe`. Follow these steps to do so:

1. Inside Visual Studio, click Project > Project Settings.
2. In the Settings window, you see a Ruby region with a “Ruby Interpreter” line. Change the value on this line to the path of `ir.exe`.  
 For example, if you installed IronRuby in `C:\IronRuby`, you must set the value to “`C:\IronRuby\Bin\ir.exe`”.
3. Save the project and press `Ctrl + F5` to execute the Ruby files.

Ruby in Steel is a commercial product that costs money. It supports Visual Studio 2005, 2008, and also machines without Visual Studio at all (uses the Visual Studio Shell).

To read more about it, try it, or buy it, visit <http://www.sapphiresteel.com/Ruby-In-Steel-Developer-Overview>.

## NetBeans

NetBeans is a free, open source IDE that supports several programming languages, along with the Ruby language. It is Java-based and can run on all operating systems that run Java applications.

For Ruby, you get code completion, naming convention warnings, a convenient project tree, and Ruby on Rails support.

Figure 4.8 shows the NetBeans window with Ruby code inside.

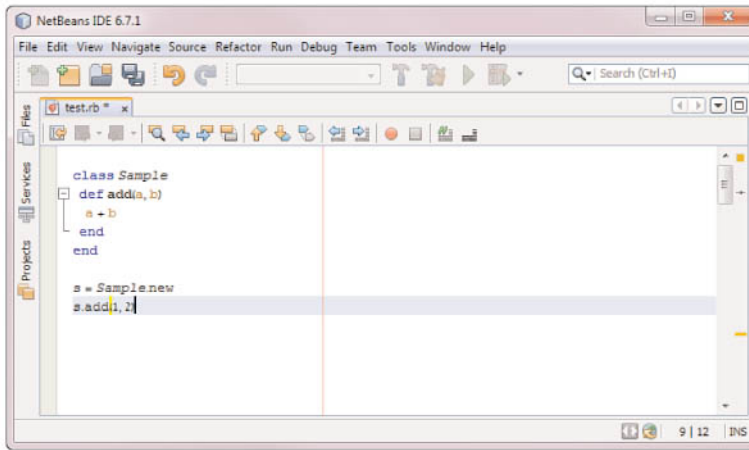


FIGURE 4.8 NetBeans IDE screenshot with Ruby code.

Unfortunately, NetBeans in its current version 6.7 doesn't support IronRuby. Therefore, you cannot run or debug IronRuby code directly from NetBeans. You must run the IronRuby file from the command prompt using `ir.exe`.

The NetBeans team plans to add IronRuby to its supported Ruby platforms in one of its next versions.

To learn more about NetBeans and download it, visit <http://www.netbeans.org>.

## RubyMine

RubyMine is a commercial Ruby IDE by JetBrains. This is one of the most advanced Ruby IDEs available and features project creation wizards, syntax highlighting, code tools (like a “surround with” function), intellisense, refactoring, and version control system integration.

Figure 4.9 shows the RubyMine interface and its intellisense pop-up.

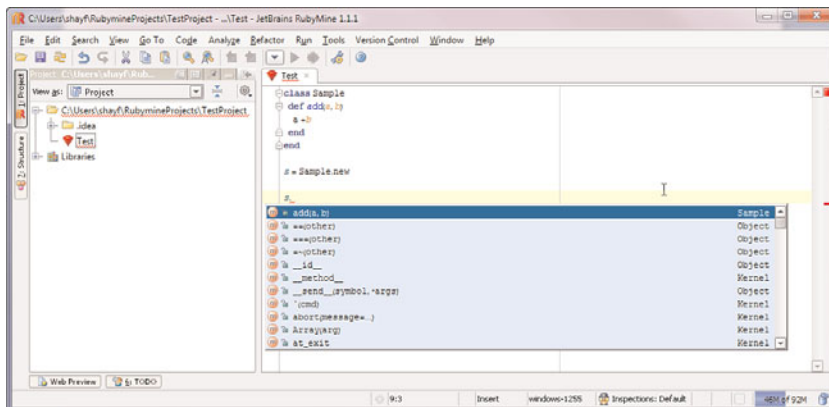


FIGURE 4.9 RubyMine IDE screenshot with intellisense.

RubyMine doesn't come with IronRuby as its Ruby interpreter, and you have to add it as one to execute files with the IronRuby interpreter directly from the interface.

Follow these steps to add IronRuby as a Ruby interpreter in RubyMine:

1. Go to File > Settings.
2. On the left, choose Ruby SDKs and Gems.
3. On the settings on the right, click the Add SDK button, which is located in the upper-right corner of the dialog.
4. On the file selector dialog that opens, navigate to <IronRuby installation folder>\Bin\ir.exe. Click OK after you select the file.
5. When you click OK on the settings form, RubyMine makes IronRuby its default interpreter.

Before you run a file, you need to modify something else. RubyMine uses configuration settings for each execution. In this configuration, the command-line arguments are sent to the interpreter. The default ones do not work with IronRuby, and you need to remove them. Follow these steps to do that:

1. Go to Run > Edit Configurations.
2. Click the Edit Defaults button, which is located in the lower-left corner of the dialog.
3. Choose Ruby on the left panel.
4. On the right, clear the text from the Ruby Arguments field.
5. Click OK on all open dialogs to save the changes.

Now you can work on and run files by using the IronRuby interpreter.

RubyMine is a commercial product that costs money. It works on Windows, Mac OS X, and Linux. You can read more about it, try it, and buy it at <http://www.jetbrains.com/ruby>.

## Others

Along with these IDEs, a lot of other great IDEs are available. Some are appropriate for bigger applications, and some for smaller applications and scripts. Most of them offer simple code completion and syntax highlighting.

Like the others, they still cannot run IronRuby directly from their interface, so you have to switch to the command prompt and use ir.exe to run the Ruby file you've been working on.

Some of these IDEs are RadRails (<http://www.aptana.com/radrails>), SciTE (<http://www.scintilla.org/SciTE.html>), and Notepad++ (<http://notepad-plus.sourceforge.net>). Search the Internet for "Ruby IDEs" to find more Ruby IDEs.



## The Power of IronRuby

As a finale to the IronRuby introduction, I want to leave you wanting more. Instead of just writing a simple Hello World application, I want you to see how great IronRuby is and how it can enhance your development work.

Ruby comes with very powerful built-in metaprogramming capabilities. One of its features is a built-in method called `method_missing` (read more about it in Chapter 6, “Ruby’s Code-Containing Structures”). The `method_missing` method catches all calls to undefined methods and lets you handle them the way you want.

In Listing 4.1, by using `method_missing`, I implement a `Recorder` class that records calls to methods and plays them back on request. Notice how easy it is to accomplish this with Ruby. Even though you might not be familiar with all the syntax elements, this code is pretty simple and straightforward.

LISTING 4.1 A Recorder Class Implementation

---

```
class Recorder
  def initialize
    @calls = []
  end
  def method_missing(method, *args, &block)
    @calls << [method, args, block]
  end
  def playback(obj)
    @calls.each do |method, args, block|
      obj.send method, *args, &block
    end
  end
end
```

---

Now that the `Recorder` class is ready, we can take advantage of it. Unlike regular Ruby code, we can take advantage of IronRuby capabilities and run the `Recorder` class with .NET code. Listing 4.2 takes advantage of the `Stack` class of the .NET Framework and runs recorded operations on it using our `Recorder` class.

LISTING 4.2 Using the Recorder Class on a CLR Object

---

```
recorder = Recorder.new
recorder.push 5.6
recorder.pop
recorder.push 1
recorder.push "IronRuby"
```

---

```
# Run the recorded calls on the CLR Stack instance
```

```
stack = System::Collections::Stack.new
recorder.playback(stack)
recorder.playback(stack)

stack.each { |x| puts x }
# Prints "IronRuby
#         1
#         IronRuby
#         1"
```

---

## Summary

After reading this chapter, you are ready to start developing with IronRuby on your computer. You have installed IronRuby and have been introduced to the tools it comes with. This chapter also covered a few options for IronRuby development environments. At the end of the chapter, you saw an IronRuby sample that gave you a taste of the great power IronRuby holds inside.

We now delve further into the Ruby language and examine the fundamentals of IronRuby, and then you learn how to use IronRuby with the different frameworks of Ruby and .NET, all on the road to becoming an IronRuby master.

*This page intentionally left blank*

# PART II

## The Ruby Language

### IN THIS PART

|           |                                   |     |
|-----------|-----------------------------------|-----|
| CHAPTER 5 | Ruby Basics                       | 43  |
| CHAPTER 6 | Ruby's Code-Containing Structures | 87  |
| CHAPTER 7 | The Standard Library              | 131 |
| CHAPTER 8 | Advanced Ruby                     | 161 |

*This page intentionally left blank*

# CHAPTER 5

## Ruby Basics

### IN THIS CHAPTER

- ▶ Basic Syntax
- ▶ Hello World
- ▶ Variables
- ▶ Control Structures
- ▶ Exception Handling

**I**ronRuby is, first and foremost, an implementation of the Ruby language. Therefore, to know IronRuby you first need to get familiar with its foundation—the Ruby language.

This chapter can help you to get to know Ruby. It covers the basic concepts of the language including basic syntax and common programming tasks. After reading this chapter, you will be able to read and write simple Ruby applications.

## Basic Syntax

Every programming language consists of a few building blocks that you must know to use all other language parts that are built on top of them. The Ruby language is no exception. Hence, before we delve into the world of Ruby, we get familiar with its basic syntax.

## Comments

Comments are important. They make your application clearer to other developers, and to yourself. Ruby features two types of comments: single line and multiline.

The single-line comments start with a sharp character (#):

```
# This is a comment
num = 5 # Comments can start at the end of a code line
➡as well
```

The multiline comments start with `=begin` and end with `=end`:

```
=begin
  Here you can write
  as many lines as you wish
  and they will all be ignored by the compiler.
=end
```

### NO MIDDLE-LINE COMMENTS

Ruby does not have a way to write comments on the middle of the line, like `/* */` in C++ or C#.

## Setting Values to Variables

Ruby is a highly dynamic language and does not require types to be defined explicitly. Furthermore, there is no need to declare variables. The approach is the opposite: There are ways to determine whether a variable has already been declared.

Setting a value to a variable can create the variable if needed. The operation is done by writing the variable name and its value in the following way:

```
my_variable = 1
another_variable = "Hello"
```

### USE OR LOOSE SYNTAX ELEMENTS

Ruby allows you to use or lose certain syntax characters. One of them is the semicolon. For example

```
my_variable = 1
```

is equivalent to

```
my_variable = 1;
```

The semicolon becomes handy when you want to write several statements in the same line:

```
my_variable = 1; my_variable += 1; print my_variable;
```

Be aware that the semicolon is almost never used in Ruby and should be avoided in general (to keep the code clear and readable).

Ruby also allows parallel assignments. This means that in a single line you can assign values to multiple parameters:

```
one, two = 1, 2 # one now contains 1 and two contains 2
```

A special case of parallel assignment is arrays:

```
one, two = [1, 2]
```

This line is identical to the preceding one; it assigns 1 to one and 2 to two. However, the next line is different:

```
one, two = [1, 2], 3
```

After this line is executed, one contains the array [1,2], and two contains 3.

To understand this behavior, think of the assignment values as an array. When you write `one, two = 1, 2`, it is actually `one, two = [1, 2]`. Consequently, `one, two = [1, 2], 3` is actually `one, two = [[1, 2], 3]`.

You can bypass this using the `*` operator. When the `*` operator is put before an array, it flattens it so that `*[1, 2]` becomes `1, 2`:

```
one, two, three = 1, *[2, 3] # one = 1, two = 2, three = 3
```

### THE \* OPERATOR ALLOWED POSITION

The `*` operator is allowed only on the last value of the assignment. For example, the next statement results in a syntax error:

```
one, two, three = *[1, 2], 3
```

---

## Calling Methods

Calling methods in Ruby can be done with or without parentheses. This allows method invocations to look like statements, which make it very human readable. For instance, the next two lines are equivalent:

```
method_name(param1, param2)
method_name param1, param2
```

### WHITESPACES AND PARENTHESES

When you do want to use parentheses, you must make sure that the opening bracket follows the method name, with no whitespaces between them.

The next two lines are entirely different:

```
method_name(1+2)+5
method_name (1+2)+5
```

The first line will pass 3 to the method and add 5 to its result. The second line sends 8 as a parameter to the method.

---



An issue to notice is calling two methods one after another without parentheses. This confuses the Ruby interpreter and results with a warning or even an error in some cases.

For example, the next line

```
puts add 1, 3
```

can be interpreted in multiple ways:

```
puts add(1, 3)
# Or
puts add(1), 3
# Or maybe even
puts; add(1, 3)
```

To resolve this behavior, the most obvious solution is to put parentheses around the parameters passed to the `add` method. Another way to resolve the issue is by surrounding `add` with parentheses, along with a whitespace after the first method call:

```
puts (add 1, 3)
```

## Code File Structure

Ruby files have a few structure rules.

If you run your application on UNIX-like operating systems, the first line of the code should be a “shebang” comment that tells the operating system how to execute the file. This line can be omitted in Windows or Mac:

```
#!/usr/bin/ruby -w
```

The next lines should consist of `require` statements. A `require` statement allows loading other code files and compiled libraries (DLL files, for example):

```
require 'base64' # Load Ruby's base64 library
require 'otherCodeFile.rb' # Load local otherCodeFile.rb file
require 'C:\RubyFiles\MyRubyLib.rb' # Load Ruby file from a specific location
```

The `require` statements will be followed by the program code. There are no strict restrictions for how the code should appear. Nevertheless, when a code container is declared, a code hierarchy should be kept according to the next order:

```
module ModuleName
  class ClassName
    def method_name
      while condition do
        if condition then
          ... code ...
        end
      end
    end
  end
end
```

```
        end
      end
    end
  end
end
```

If the special statement `__END__` appears somewhere within the code file, everything after it is ignored and is not executed. The `__END__` statement signals to the Ruby interpreter the end of the file. It is optional; when it doesn't appear, the whole file is processed:

```
... code ...
```

```
__END__
```

We can write here everything we want and it will not be processed by the interpreter.

## Coding Standards

Ruby has somewhat different coding standards than, for example, C# or VB.Net. It is important to know these standards because Ruby code you will run into will most likely comply with them.

### Comments

Method, class, and module description comments usually follow RDoc's rules, to allow the use of `ri` with them. (The `ri` tool is introduced in Chapter 4, "Getting Started with IronRuby.")

RDoc documentation can be found at <http://rdoc.sourceforge.net/doc>.

### Variables

Variables are written all lowercase and with an underscore to separate words (for example, `my_variable`).

### Constants

Constants are written all uppercase and with an underline as word delimiter (for example, `THIS_IS_A_CONSTANT`).

### Methods

Methods are written all lowercase and with an underscore to separate words (for example, `my_method`).

### Classes and Modules

Class and module names are written using PascalCase. Therefore, there are no word delimiters, but the first letter of each word is capitalized (for example, `MyClassName`, `IAMAModule`).

## Hello World

Ruby is a very flexible language. Therefore, you write only what you need. You don't have to adhere to strict rules for indentation, special coding ceremonies, and so forth.

Writing a program that prints "Hello World!" to the screen is as simple as it sounds:

```
print "Hello World!"
```

This actually invokes Ruby's kernel method – `print` with `"Hello World!"` as its parameter.

## Variables

Variables in Ruby have implicit types. Therefore, you don't declare types; Ruby does that for you.

Even though you don't explicitly declare types, variables do have types. These types are computed during runtime. As a result, variables are type-safe and obey type restrictions. For example, the next code block ends up with an exception:

```
num = 1
str = "Hello"
mix = num + str
```

However, the following will work just fine:

```
my_variable = 1
my_variable = "Hello"
```

This code works because there is no violation of any type restriction. On the second line, `my_variable` type is changed from a numeric 1 to a textual one.

Ruby comes with a set of basic types. The base type is `Object`. Everything is an object in Ruby, and because the language is entirely object oriented, every object in Ruby inherits from the `Object` class implicitly or explicitly.

## Numbers

Ruby contains several types for numeric values. These include types for integers, floats and the standard library adds rational and complex numbers support. All numbers in Ruby derive from the `Numeric` class. Figure 5.1 presents the inheritance tree of Ruby numeric types.

All integers are of the `Integer` type. If a value is up to 31 bits, it will become a `Fixnum`. Otherwise, the `Bignum` class will be used. In case of real numbers, the `Float` class will be picked, because it has a built-in support for a floating point. Numbers in Ruby never overflow. When a `Fixnum` reaches its limit, it is converted to a `Bignum`.

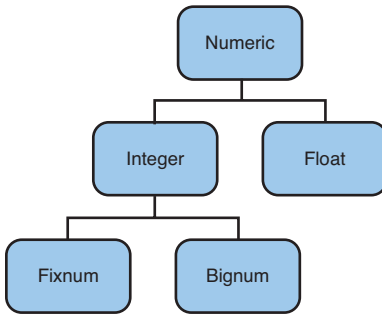


FIGURE 5.1 The inheritance tree of numeric types in Ruby.

### Integer Declaration

You can declare integers in the regular numeric way or in octal, hexadecimal, or binary way. For example, the next statements are all equivalent:

```

num_integer = 777
num_octal = 01411 # octal way, value starts with a zero
num_binary = 0b1100001001 # binary way, starts with 0b
num_hexadecimal = 0x309 # hexadecimal way, starts with 0x

```

## MAKING BIG NUMBERS MORE HUMAN READABLE

You can use an underscore to make big numbers more readable:

```

num1 = 1000000000 # A billion
num2 = 1_000_000_000 # A billion. Equivalent to the previous line

```

### Float Declaration

Float values have a floating point. This is all you need to define a float:

```

float_value = 34.89
negative_float_value = -12.35

```

### Arithmetic

All numeric types support the `+`, `-`, `*`, `/`, and `**` (power) operators. You can also use them with an equal operator. For instance, these are equivalent:

```

num = 1 + 2 # = 3

num = 1
num += 2 # = 3

```

## DIVIDING INTEGERS

When you divide two integers, the result is an integer, too. To avoid that, you can add a floating point to the values or convert them explicitly to a `Float`:

```
result = 5 / 2 # = 2
result = 5.0 / 2 # = 2.5

int1, int2 = 5, 2
result = int1.to_f / int2 # = 2.5
```

---

## Text

Text is represented in Ruby by the `String` class. The `String` class contains various methods to help you create, read, manipulate, and delete text.

### Defining Strings

There are a few different ways to define strings in Ruby, each with its own purpose.

#### Double-Quoted Strings

The most common and flexible approach is to use double-quoted strings:

```
my_string = "Hello there"
```

The double-quoted string supports several escape characters, like `\n` for a new line and `\t` for a Tab space. It can also be expended to multiple lines:

```
my_string = "Hello
there!
```

```
This is a three line string."
```

## USING A DIFFERENT STRING DELIMITER

In case you have a string with lots of quotes, you can define a different delimiter for the string. This is done using the `%q` and `%Q` statements. The next character is the delimiter, and the string limiter continues until this character is found again. The difference between `%q` and `%Q` is that `%q` generates a single-quoted string and `%Q` generates a double-quoted string:

```
my_string = %q^I've seen enough of the chef's food!^
# Equivalent to: 'I've seen enough of the chef's food!'

my_string = %Q~The "car" is actually a horse~
# Equivalent to: "The \"car\" is actually a horse"
```

---

**String.new**

Using the `String.new` method achieves the same as the double-quoted string:

```
my_string = String.new "Hello" # = A string containing "Hello"
```

Without arguments, an empty string is created:

```
my_string = String.new # = An empty string
```

**Single-Quoted Strings**

Another way to create strings is to use a single-quoted string:

```
my_string = 'A \'single-quoted string\'' # = A 'single-quoted string'
```

The single-quoted string is rarely used because it is not as flexible as the double-quoted string. For example, it does not support most of the escape characters (only single quote and backslash). However, using the single-quoted string can improve performance, although this should be your last resort for that.

**A CHARACTER'S NUMERIC VALUE**

To get a character's numeric value, you can write a question mark followed immediately by the character:

```
num = ?A # = 65
```

```
num = ?& # = 38
```

**Here Documents**

Here documents, also known as *heredocs*, are suitable for long text blocks. You define a delimiter, which when it appears again, it signals the end of the text chunk. A `<<` starts a here document, and the delimiter then follows. Then the text and the delimiter to signal the end of the text block follow:

```
my_string = <<DOC
Hello there! # This is part of the text... Can't put comments inside heredocs
This is the second line.
DOC
```

```
my_string = <<'END OF MY STRING'
I'm writing here whatever I want.
Yes, everything goes!
Hallelujah!
END OF MY STRING
```

**Expression Interpolation**

This is a way to format strings. Expression interpolation means you can insert Ruby code into strings. The way to do that is to insert `#{expression}` into the string:

```
num = 5
string = "#{num}+1 = #{num+1}" # = "5+1 = 6"
```

This also works in here documents and can be handy (for example, with XML blocks):

```
value = 6
string = <<END
<root>
  <value>#{value}</value>
</root>
END
```

**printf**

Similar to the expression interpolation technique, `printf` allows formatting strings. `printf` gives more control over the format of the variables in terms of width, floating-point precision, and more.

The `printf` method receives the string with format sequences and the variables afterward. This is different from string interpolation, where the variables are inserted into the formatted string.

A format sequence in the string consists of a percent sign (%) followed by option flags and width and precision indicators. The format sequence ends with a field type indicator.

The following code samples demonstrate the most common uses of `printf`. To read more about `printf` and its different flags, refer to [www.ruby-doc.org/core/classes/Kernel.html#M005984](http://www.ruby-doc.org/core/classes/Kernel.html#M005984).

```
# Show only 2 digits after the floating point:
printf("%.2f", 12.12345) # = 12.12
```

```
# Add leading zeros if the number is smaller than 3 digits:
printf("%03d", 12) # = 012
```

```
# Enforce at least 5 characters to be shown (zeros will be added if needed)
# and show only 2 digits after the floating point:
printf("%05.2f", 1.12345) # = 01.12
```

```
# Show at least 10 characters and keep the original string on the left
printf("%-10s", "IronRuby") # = "IronRuby "
```

```
# Use the same value multiple times
printf("%1$s %2$s, %2$s %1$s, %2$s %2$s %2$s", "hello", "Ruby!")
# = "hello Ruby!, Ruby! hello, Ruby! Ruby! Ruby!"
```

`sprintf` is identical to `printf` in terms of arguments and flags. The only difference is that `sprintf` returns a new generated string and `printf` prints it immediately to the screen:

```
str = sprintf("%-10s", "IronRuby") # str now equals "IronRuby "
```

The `String` class offers another way of doing `sprintf` operations: the `%` operator. The percent sign operator does exactly the same as `sprintf`. It assumes that the string before the operator is the format string and the variables after the operator are the values.

For example, the following line

```
str = sprintf("%-10s", "IronRuby")
```

is equivalent to this:

```
str = "%-10s" % "IronRuby"
```

When the `%` operator is sent with multiple values, use square brackets to wrap them as an array first:

```
str = "%s %s !!!" % ["IronRuby", "Unleashed"] # str now equals "IronRuby Unleashed !!!"
```

### Accessing Strings

You can access substrings within Ruby strings using the array-index operator, `[]`. The substrings are both readable and writable. Using an integer, you can access single characters within the string:

```
my_string = "Hello There"
my_string[0] # = 72 - the first letter ASCII code
my_string[my_string.length-1] # = 101 - the last letter ASCII code
my_string[-1] # Another way to get the last character = 101
my_string[0] = "j" # Now the string is "jello There"
my_string[100] # = nil. Index doesn't exist.
```

The way to access whole substrings within the string is to use two integers. The first indicates the starting position, and the second indicates the length of the substring:

```
my_string = "Hello There"
my_string[0,2] # = "He"
my_string[my_string.length-1, 5] # = "e" (the index can exceed the string length)
my_string[-5, 5] # = "There"
my_string[-5,5] = "Ruby" # The string is now "Hello Ruby"
```

Another way to access substrings is by using ranges. A range in Ruby is two integers separated with two dots. The integers represent two indexes in the string. This is an important difference from the previous way: The two integers do not represent an index and a length; they stand for two indexes:



```
my_string = "Hello There"
my_string[3..4] # = "lo"
my_string[6..my_string.length] # = "There"
my_string[1..3] = "ipp" # The string is now "Hippo There"
```

### String Class Methods

The String class features several methods that return the length of the string, go through the string characters, and more.

length and size are identical and both return the length of the string:

```
"IronRuby Unleashed".length # = 18
```

strip removes whitespaces from the string beginning and end. strip creates a new string and returns it. Use strip! to change the original string:

```
"    IronRuby Unleashed    ".strip # = "IronRuby Unleashed"
```

### STRIP ONE SIDE ONLY

If you want to remove whitespaces from only the beginning of the string, use lstrip. If you want to remove them from the end, userstrip:

```
"    IronRuby Unleashed    ".lstrip # = "IronRuby Unleashed    "
"    IronRuby Unleashed    ".rstrip # = "    IronRuby Unleashed"
```

include? returns true if the given string exists within the string:

```
"IronRuby Unleashed".include?("Ruby") # = true
```

each loops over the string content by a given separator:

```
str = "IronRuby Rocks!"
str.each(" ") { |x| print "->#{x}<-" } # prints "->IronRuby <->Rocks!<-"
str.each("R") { |x| print "->#{x}<-" } # prints "->IronR<->uby R<->ocks!<-"
```

## Arrays

Arrays are indexed collections. Every value can be accessed by its position in the array. Arrays in Ruby grow automatically when needed and can hold objects of any type.

### Defining Arrays

Ruby arrays can be defined and constructed in multiple ways. The most straightforward way to use square brackets follows:

```
my_array = [ "Hello", 5, Time.now, 5**7 ]
```

The preceding code constructs an array with four items of different types.

### USING A DIFFERENT VALUE DELIMITER

If you have multiple string values and each value doesn't have spaces, you can use the %w or %W operators. These allow you to define the delimiter of the array. Similar to the String's %q and %Q, %w generates single-quoted values, and %W generates double-quoted values. After the delimiter is set, the values inside are separated by spaces:

```
my_array = %w-A B C 5- # Same as ['A', 'B', 'C', '5']
```

```
my_array = %W-"Iron Ruby" V1~ # Same as ["Iron", "Ruby", "V1"]
```

Another way to generate an array is by using Array.new. This method enables you to create an array with an initial number of positions, set initial values to them, and copy a different array:

```
array1 = Array.new # = an empty array
```

```
array1 = Array.new(5) # = [ nil, nil, nil, nil, nil ]
```

```
array1 = Array.new(5, "hey") # = [ "hey", "hey", "hey", "hey", "hey" ]
```

```
array1 = Array.new(5) { |i| 5 * i } # = [ 0, 5, 10, 15, 20 ]
```

```
array2 = Array.new(array1) # creates a clone of array1
```

The next approach can help you construct arrays. After you define an array, you can add items to it using the append (<<) operator:

```
my_array = Array.new # = an empty array
```

```
my_array << "Hello" # the array is now [ "Hello" ]
```

```
my_array << 5 << Time.now # The array is now [ "Hello", 5, Time.now ]
```

### Accessing Arrays

Accessing array items is similar to accessing substrings within Ruby strings. After an item or a range of items is accessed, it can be modified or deleted. The similarity between all ways to access arrays is that all of them use the square brackets operator, [].

The first way to access an array item is by using its index. Write the index and receive the item. You can also use negative numbers, which indicate positions from last to first:

```
my_array = [ 1, 2, 3, 4, 5 ]
```

```
my_array[0] # = 1
```

```
my_array[-4] # = 2
```

```
my_array[10] # = nil. Index doesn't exist.
```

Figure 5.2 presents the possible indexes to access array elements.

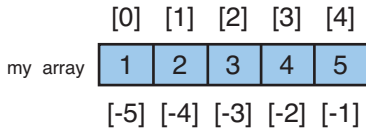


FIGURE 5.2 The available indexes to access array items.

The next way uses two integers. The first indicates the starting index, and the second specifies the number of elements to get. The number of elements can exceed the array length:

```
my_array = [ 1, 2, 3, 4, 5 ]
my_array[0,1] # = 1
my_array[1, 2] # = [2, 3]
my_array[3, 100] # = [4, 5]
my_array[1, 2] = "Hey" # The array is now [ 1, "Hey", 4, 5 ]
```

Another way to access array parts is by using ranges. The range defines a starting index and an ending index:

```
my_array = [ 1, 2, 3, 4, 5 ]
my_array[0..0] # = 1
my_array[2..4] # = [3, 4, 5]

my_array[2..3] = [6, 7] # The array is now [ 1, 2, 6, 7, 5 ]
my_array[2..3] = [] # deletes positions 2 and 3 = [ 1, 2, 5 ]
```

One more way to access the array elements is by using the `each` method. The `each` method sends every array element to the given code block. This way you can move through all the array items and process them:

```
my_array = [ 1, 2, 3, 4, 5 ]
# Print each element in a new line
my_array.each { |element| puts element }
```

### Array Class Methods

All arrays in Ruby are of type `Array`. `Array` is a class in Ruby and has several methods that you can use on every array you have.

I'll introduce you here with some of the most usable methods of the `Array` class. For a complete reference, check out the `ruby-doc` website: <http://www.ruby-doc.org/core/classes/Array.html>.

`empty?` returns true if the array is empty:

```
[].empty? # = true
[ 1, 2, 3 ].empty? # = false
```

`first` and `last` return the first and last elements of the array:

```
my_array = [ 1, 2, 3, 4, 5 ]
my_array.first # = 1
my_array.last # = 5
```

`delete` removes all elements that match the given parameter:

```
my_array = [ 1, 2, 1, 1, 5 ]
my_array.delete(1) # the array is now [2, 5]
```

`select` returns all array elements that the given block returns true for:

```
my_array = [ 9, 7, 1, 4, 12, 5 ]
my_array.select { |element| element > 6 } # = [ 9, 7, 12 ]
```

## Hashes

Hashes are similar to arrays, with one big difference: Instead of indexes, a hash maps keys to values. A key is an object itself, and when given, the corresponding value will be retrieved.

### Defining Hashes

To define a hash, you need to set the key and the value for the element. In case the key already exists, its value will be replaced.

One way to define a hash is via the `Hash.new` method. After you construct an empty hash, you add elements to it as follows:

```
hash = Hash.new # constructs an empty hash
# Add an element with key="country" and value="USA"
hash["country"] = "USA"
# Add an element with key=1776 and value= "Declaration of Independence"
hash[1776] = "Declaration of Independence"
```

Another way to define a hash is by declaring its key-value pairs between curly brackets. The key and the value should be separated by a `=>` symbol:

```
# Generate a hash with a, b, c as the keys and their ASCII codes as the values
my_hash = { "a" => ?a, "b" => ?b, "c" => ?c }
```

**Using Symbols**

Symbols in Ruby are strings with a colon before them:

```
:hello
:"there"
:'I am a symbol'
```

Symbols are used as keys in various places across the language. It is preferred to use symbols as keys because their comparison is much faster.

To use symbols in hashes, you just set them as the keys:

```
my_hash = { :a => ?a, :b => ?b, :c => ?c }
```

**Accessing Hashes**

You can access hash values by passing a key between square brackets:

```
my_hash = { :a => ?a, :b => ?b, :c => ?c }
my_hash[:a] # => ?a

my_hash = { "First Name" => "Shay", "Last Name" => "Friedman" }
my_hash["First Name"] # => "Shay"
```

Another way to access the pairs in the hash is to use the each method:

```
my_hash = { :a => ?a, :b => ?b, :c => ?c }
my_hash.each { |key, value| puts "#{key} = #{value}" }
```

This code outputs the following:

```
a = 97
b = 98
c = 99
```

**Hash Class Methods**

The Hash class, unlike the Array class, does not have index-related methods. Instead, it has unique methods of its own to support the key-value pair mechanism.

For a complete reference on the Hash class, visit the ruby-doc website: <http://www.ruby-doc.org/core/classes/Hash.html>.

`has_key?` finds out whether a given key exists in the hash:

```
my_hash = { :a => ?a, :b => ?b, :c => ?c }
my_hash.has_key?(:a) # => true
my_hash.has_key?("Hey there") # => false
```

`has_value?` determines whether a given value exists within the hash:

```
my_hash = { :a => ?a, :b => ?b, :c => ?c }
my_hash.has_value?(?b) # = true
my_hash.has_value?(?k) # = false
```

`select` returns all array elements that the given block returns true for:

```
my_hash = { :a => ?a, :b => ?b, :c => ?c }
my_hash.select { |key, value| value > ?b } # = { :c => ?c }
```

## Ranges

A range in Ruby is a series of values from one object to another that are represented as one single object. Ranges are declared by writing the start object, two or three dots, and then the end object. Two dots means that the end object is included within the range. Three dots means that the end object will be excluded from the range:

```
1..5 # = integers between 1 and 5
1.0...5.0 # = all numbers between 1 to 5, excluding 5.
```

Ranges can also be used with strings:

```
"a".."d" # = "a", "b", "c", "d"
"AAA".."BBB" # all possible combinations between "AAA" and "BBB"
# "AAA", "AAB", ..., "ABA", "ABB", ..., "ART", "ARU", ..., "BAZ", "BBA", "BBB"
```

### CONVERT A RANGE TO AN ARRAY

You can convert a range to an array using the `to_a` method:

```
my_array = ("AAA".."BBB").to_a
```

When a range is defined, you can loop over its elements using the `each` method:

```
my_range = "AAA".."BBB"
# Print all range values. Each in a new row.
my_range.each { |elem| puts elem }
```

Notice that not all ranges can be iterated over. For example, you can't iterate over Float ranges because that would be an infinite number of values.

You can also check whether a given value is a member of the range by using the `member?` and `include?` methods or the `===` operator, which all do exactly the same thing:

```
my_range = 1..50
my_range.member?(4) # = true
my_range.member?(100) # = false
my_range.include?("Hey") # = false
my_range === 40 # = true
my_range === 55 # = false
```

Custom objects can also be made available for “ranging” using the `succ` method. We discuss this in the next chapter.

## Booleans

Ruby does not have a Boolean type. Instead, it has a type for true, `TrueClass`, and a type for false, `FalseClass`. The keywords `true` and `false` actually return a singleton instance of the `TrueClass` and the `FalseClass`, accordingly.

When a Boolean value is required, `false` and `nil` result, because `false` and any other value results in `true`:

```
# This will print "Hey!" to the screen
num = 1
if (num)
  puts "Hey!"
end

# This will not print "Hey!"
num = nil
if (num)
  puts "Hey!"
end
```

## Regular Expressions

Regular expressions are “native” to the Ruby language and are used in various scenarios. A `regex` is a textual pattern that can be verified upon a given string. The `regex` can also be used to parse text and retrieve parts from it.

### Defining Regular Expressions

A regular expression in Ruby is declared between slashes:

```
regex = /^(Iron)?Ruby$/
```

This regular expression accepts only two strings: "Ruby" and "IronRuby".

A regular expression can be followed by various flags that affect the pattern-matching process:

To use these flags, add them to the end of the regular expression:

```
regex = /^(Iron)?Ruby$/i
```

Now this pattern accepts strings such as "ironruby", "RUBY", and so on.

Another way to construct a regex is by using the `Regexp.new` method:

```
Regexp.new("[a-z]*") # = /[a-z]*/
Regexp.new("[a-z]*", Regexp::IGNORECASE) # = /[a-z]*/i
Regexp.new("[a-z]*", Regexp::MULTILINE | Regexp::IGNORECASE) # = /[a-z]*/mi
```

Table 5.1 contains all possible regular expressions flags.

TABLE 5.1 Regular Expression Flags

| Flag | Description   |
|------|---|
| I    | Ignore case.  |
| M    | Multiline mode. Treats newline characters like regular characters.                    |
| X    | Allow whitespaces and comments within the regular expression.                         |
| O    | Perform string interpolation only the first time the regular expression is evaluated. |
| U    | Define the regular expression encoding to Unicode.                                    |
| N    | Define the regular expression encoding to none, which is ASCII.                       |

### Using Regular Expressions

To match a regex pattern against a string, you can use the `==` operator. This operator returns the index of the first match or `nil` if none is found:

```
regex = /(Iron)?Ruby/i
"IronRUBY" == regex # = 0
"this book is about ironruby" == regex # = 19
"Hello there" == regex # = nil
```

You can also retrieve the matches after a successful pattern matching. After a successful execution of `==`, a thread-local and method-local variable are available to you: `$~`. This variable holds a `MatchData` object that you can inspect and thus get information about the found matches:

```
regex = /(Iron)?Ruby/i
"we love IronRuby" == regex
```



```
puts $~[0] # = "IronRuby". The whole match.
puts $~[1] # = "Iron". First subpattern match.
puts $~.begin(1) # = 8. The start position of the first subpattern.
puts $~.end(0) # = 16. The end position of the whole match.
```

## Date and Time

Ruby has a built-in class to handle dates and times: the `Time` class. This class features several methods and attributes related to dates and time.

`now` returns a `Time` object representing the current date and time. `new` is identical to `now`:

```
Time.now
Time.new
```

`local`, `utc` and `gmt`, which is identical to `utc`, construct a new `Time` object from the given parameters. `local` creates the time in the local time zone and `utc` in the GMT time zone:

```
Time.local(2009, 9, 6, 15, 0, 50) # = September 6 2009, 3pm and 50 seconds
```

`year`, `month`, `day`, `hour`, `min`, and `sec` return their related part of the time:

```
t = Time.local(2009, 9, 6, 15, 0, 50)
t.year # = 2009
t.month # = 9
t.day # = 6
t.hour # = 15
t.min # = 0
t.sec # = 50
```

`strftime` converts the `Time` object to string in a given format:

```
t = Time.local(2009, 9, 6, 15, 0, 50)
t.strftime("%B %d %Y %H:%M") # = "September 06 2009 15:00"
```

Table 5.2 contains all possible format directives that can be used on the `strftime` method.

TABLE 5.2 Format Directives

| Directive | Description                                  |
|-----------|--|
| %a        | Abbreviated weekday name ("Sun")             |
| %A        | Full weekday name                            |
| %b        | Abbreviated month name ("Jan")               |
| %B        | Full month name                              |
| %c        | Preferred local date and time representation |

TABLE 5.2 Format Directives

| Directive | Description  |
|-----------|--|
| %d        | Day of the month   |
| %H        | Hour of the day, 24-hour clock                                   |
| %I        | Hour of the day, 12-hour clock                                   |
| %j        | Day of the year  |
| %m        | Month  |
| %M        | Minute of the hour   |
| %p        | “AM” or “PM”   |
| %S        | Second of the minute   |
| %U        | Week number of the current year, starting with the first Sunday. |
| %W        | Week number of the current year, starting with the first Monday. |
| %w        | Day of the week (Sunday = 0)                                     |
| %x        | Preferred representation for the date alone, no time             |
| %X        | Preferred representation for the time alone, no date             |
| %y        | Year without a century (00..99)                                  |
| %Y        | Year with century  |
| %Z        | Time zone name   |
| %%        | A % character  |

For a complete reference of the `Time` class, refer to <http://www.ruby-doc.org/core/classes/Time.html>.

## Constants

Constants in Ruby are similar to variables. The language doesn't even restrict constants to stay with the same value. The only difference is that the Ruby interpreter generates a warning when the value is changed.

To indicate to Ruby that a variable is a constant, you just name it with a capital letter. For example, `helloWorld` is a regular variable, whereas `HelloWorld` is a constant.

Declaring and using constants is just like variables then:

```
BOOK_NAME = "IronRuby Unleashed"
puts "You are reading #{BOOK_NAME}"
```

If you try to set a new value to `BOOK_NAME`, you receive a warning: Already initialized constant `BOOK_NAME`.

Constants names can also be attached to a class or module object. This is done using the double-colon operator (`::`), as follows:

```
class Books
end

# A constant attached to a class named Books
Books::IRON_RUBY = "IronRuby Unleashed"
```

This is equivalent to the following:

```
class Books
  IRON_RUBY = "IronRuby Unleashed"
end
```

Constants appear in the global context. You can access them from outside their original scope. For example, using the `Books` class `IRON_RUBY` constant from outside is simple:

```
class Books
  IRON_RUBY = "IronRuby Unleashed"
end

puts Books::IRON_RUBY
```

## Control Structures

Just like any other programming language, Ruby has control structures (for example, different conditions, loops, and enumeration possibilities). Even though control structures are quite “rock solid” across different languages, Ruby manages to provide new and helpful statements to make the code more readable and maintainable.

### Conditions

#### Comparison Operators

Before we start talking about conditions, let’s take a look at the comparison operators Ruby offers. Table 5.3 contains the available comparison operators of the Ruby language.

#### BOOLEAN OPERATOR’S PRECEDENCE

Notice the Boolean operator’s precedence. This is an important thing to notice because you might experience unexpected behavior if you are not familiar with those.

Consider the next statement:

```
1 || 2 && false
```

Because `&&` has a higher precedence than the rest of the Boolean operators, it is performed first. So the preceding statement is identical to the following:

```
1 || (2 && false)
```

Therefore, the result of the statement is 1.

The named Boolean operators, `and` and `or`, have the lowest precedence. (Both have the same precedence, though.) Using `and` and `or` only ensures that the Boolean statement is evaluated from left to right. This is why the next statement returns `false`:

```
1 or 2 and false
```

TABLE 5.3 Comparison Operators

| Operator                               | Description   |
|--|---|
| <code>==</code>                        | Equal to.   |
| <code>!=</code>                        | Not equal to.   |
| <code>&lt;</code> , <code>&lt;=</code> | Less than / less than or equal to.  |
| <code>&gt;</code> , <code>&gt;=</code> | Greater than / greater than or equal to.  |
| <code>&lt;=&gt;</code>                 | General comparison. If the operand on the left is less than the operand on the right, -1 is returned. If the operand on the left is greater, 1 is returned. In case the operands are equal, 0 is returned. If the operands are not comparable together, <code>nil</code> is returned. |
| <code>==~</code>                       | Pattern match (see regular expressions).  |
| <code>!~</code>                        | No pattern match.   |
| <code>===</code>                       | Case equality. Mostly used implicitly in case statements. It has different implementations. For example, for ranges it checks for membership, and for regular expressions it tries to pattern match.  |
| <code>&amp;&amp;</code>                | Boolean AND (highest precedence).   |
| <code>  </code>                        | Boolean OR (high precedence).   |
| <code>AND</code>                       | Boolean AND (low precedence).   |
| <code>OR</code>                        | Boolean OR (low precedence).  |

## If

`if` is the base condition of any language, and Ruby is no exception. The simple syntax of `if` is as follows:

```
if condition
  ...code...
end
```

For example:

```
x = 5
if x > 2
  x = 7
end
```

You can also use `then` to strictly define the end of the condition:

```
if condition then
  ...code...
end
```

For instance:

```
x = 5
if x > 2 then
  x = 7
end
```

`if` can be used on a single line, too. A single-line `if` has two available syntaxes. The first is the usual way, where you must use the `then` keyword so that it is clear where the condition ends:

```
x = "Ruby"
if x == "Ruby" then x = "IronRuby" end
```

The second way to write a single-line `if` statement is by writing the code block first and the `if` condition afterward. This way is also called an *if modifier*. It makes the statement sound entirely human when said out loud (try it):

```
language = "IronRuby"
print "IronRuby Rocks!" if language.eql?("IronRuby")
```

The `if` syntax also supports `elsif` and `else`. `if` must be declared as the first condition. `elsif` can appear multiple times afterward, with more conditions. And `else`, when it appears, must be the last of the bunch. The `end` keyword appears only at the end of the conditional statement:

```
person = { :FirstName => "Shay", :LastName => "Friedman" }

if person[:FirstName] == "John"
  puts "You're John!"
elsif person[:LastName] == "Doe"
  puts "You're a Doe, but not John!"
elsif person.length > 2 then # then can be used on elsif too
```

```
puts "We have more than just first and last name!"
else
  puts "Your name is #{person[:FirstName]} #{person[:LastName]}"
end
```

## IF AS AN EXPRESSION

Just like everything else in Ruby, `if` is an expression. This allows you to return a value from an `if` statement straight into a variable:

```
x = 6
day_name = if x == 1 then "Monday"
            elsif x == 2 then "Tuesday"
            elsif x == 3 then "Wednesday"
            elsif x == 4 then "Thursday"
            elsif x == 5 then "Friday"
            else "Weekend!" end
```

---

### Unless

`unless` is the opposite of `if`. It specifies a condition that passes in case it returns `false` or `nil`. The `unless` statement structure is identical to `if`'s, except for one difference: `elsif` statements are not allowed.

```
x = 5
unless x.nil? # identical to: if not (x == nil)
  puts x
end
```

```
unless x < 5 then puts x end #identical to: if x >= 5 then...
```

```
unless x < 5
  puts x
else
  puts "Smaller than 5"
end
```

```
# Also support modifier syntax:
puts x unless x < 5
```

### Case

The `case` statement is a powerful control structure in Ruby. It has various usages, as discussed in just a moment. The `case` statement consists of `case`, `when`, `else`, and `end` keywords. `case` always starts the expression, optionally following an associated expression,

and then several `when` clauses appear with conditions, and `else` optionally ends the structure. The `end` keyword closes the statement:

```
case
  when condition
    ...code...
  when condition
    ...code...
  else
    ...code...
end
```

Only the first `when` clause whose condition results in `true` is processed. If no `when` condition is `true` and an `else` clause exists, the `else` code block is executed. Otherwise, no code is executed and the `case` expression returns `nil`.

The first usage of the `case` statement is as a replacement for the `if-elsif-else` syntax:

```
x = 5
case
  when x == 5
    puts "It's a five"
  when x > 5 then puts "Bigger than five" # Single line is available using then
  else
    puts "Smaller than five"
end
```

This is equivalent to the following:

```
if x == 5 then puts "It's a five"
elsif x > 5 then puts "Bigger than five"
else puts "Smaller than five"
end
```

The second way to use `case` is by using an associated expression. The expression comes right after the `case` keyword and is evaluated only once. Then, every `when` condition defines a value the expression should be equal to. The comparison is done using the `===` operator, which is not the pure comparison operator. For example, for ranges it checks for membership in the range, for classes it checks whether the expression is an instance of the given class, and for regular expressions it tries to match the string to the pattern. This mechanism gives `case` its real power, simplifying the code and its writing and maintenance processes:

```
day_num = 6
case day_num
```

```
when 1 then puts "Monday"
when 2 then puts "Tuesday"
when 3 then puts "Wednesday"
when 4 then puts "Thursday"
when 5 then puts "Friday"
when 6 || 7 then puts "Weekend" # an Or (||) operator is valid here
else puts "Not valid"
end
```

```
# Range can be used as well
case day_num
  when 1..5 then puts "Work"
  when 6..7 then puts "Rest"
  else puts "Hmmm..."
end
```

```
# Act according to variable type
case x
  when String
    puts "text"
  when Numeric
    puts "number"
  when Array
    puts "array"
end
```

Because every statement in Ruby is an expression, the case statement can also return a value:

```
action = case day_num
  when 1..5 then "Work"
  when 6..7 then "Rest"
  else "Hmmm..."
end
```

### The Ternary Operator

Just like in various other languages, you can use `?` and `:` to write a shorthand version of if-else statements:

```
day_num = 3
action = day_num > 5 ? "Rest" : "Work"
```



## Loops

Ruby offers three kinds of simple loops: `while`, `until`, and `for`. Apart from these, enumerable objects are used extensively in the Ruby language, even more than the plain simple loops.

### **while**

Similar to other programming languages, `while` in Ruby defines a code block that is executed repetitively as long as the loop condition turns out `false` or `nil`. The syntax is as follows:

```
while condition do
  ...code...
end
```

`do` may be omitted if the condition ends with a newline character, just like `if`'s `then` keyword:

```
x = 5
while x > 2 and x < 10
  puts x
  x += 1
end
```

```
while x <= 100 do x *= 5 end
```

`while` in Ruby can also be used as a modifier and be written after the loop code, just like the `if` statement:

```
x = 5
x -= 1 while x > 1
```

**Inverted while** The `while` statement can also be written in an inverted way:

```
begin
  ...code...
end while condition
```

When the statement is written this way, the associated code block is executed at least once. Only after the first iteration is the loop condition tested.

### **until**

`until` is the opposite of `while`. Its syntax is identical to `while`'s, and the loop repeats until the condition becomes `true`. It can also be used as a modifier:

```

x = 5
until x > 100
  x = x**2
  puts x
end

until x == 5 do x = Math.sqrt(x); puts x; end

my_array = [ 1, 2, 3 ]
puts my_array.pop until my_array.empty?

```

**Inverted until** Similar to the inverted while syntax, until can also be declared after the loop-associated code block:

```

begin
  ...code...
end until condition

```

### for

The for loop iterates through the elements of a given object. On each loop iteration, the current element is assigned to the loop variable, and the loop code is executed. The for syntax includes the in keyword, too:

```

for loop_variable in enumerable_object
  ...code...
end

```

As you may have noticed, for in Ruby is more similar to the foreach loop in other languages than it is to the regular for loop. This is correct. The “regular” for behavior can be achieved in other ways, which are explained in Chapter 6, “Ruby’s Code-Containing Structures.”

You can use commas to define multiple variables as loop variables to get more than a single item at a time. If there are more variables than values, the spare variables are assigned nil:

```

my_array = [ 1, 2, 3 ]
for num in my_array
  print num
end
# Output: 123

```

```

for a,b in my_array
  print a
  print b
end

```

```

end
# Output: 123nil

my_array = [ ['a','b','c'], ['d','e','f'], ['g','h','i'] ]
for a,b,c in my_array
  print a
  print b
  print c
end
# Output: abcdefghi

my_hash = { :a => ?a, :b => ?b }
for key,value in my_hash
  puts "#{key} = #{value}"
end
# Output:
# a = 97
# b = 98

```

### loop

Ruby has another interesting loop directive: `loop`. The `loop` loop has no condition, and it intends to loop indefinitely. The only way to exit a `loop` loop is by using the `break` keyword, which is introduced later in this chapter.

You can define a `loop` loop by calling the `loop` keyword and declaring its associated code block afterward:

```

# With do-end:
loop do
  ...code...
end

# Or with curly brackets:
loop {
  ...code...
}

```

### Enumerable Objects

Enumerable objects are objects that can be iterated throughout. All of Ruby's base types are enumerable. Therefore, you don't need `while`, `until`, or `for` loops to go through them. Enumerable objects also help us achieve the familiar `for` loop behavior we discussed earlier.

Before I describe enumerable objects, I want to introduce you to Ruby's blocks. A block in Ruby is an anonymous method—a method with statements and return values but without a name. There are two ways to write a code block that matches a specific specification.

The first way is a single-line block. This block is surrounded with curly brackets and optionally starts with a definition of the expected parameters between vertical bars (`|`). After the parameters have been defined, they can be used inside the code block:

```
{ puts "Hey!" } # no parameters defined
{ |x, y| x = x + y } # expects 2 parameters: x and y.
```

The second way is the multiline approach. Its syntax is started with a `do`, and then optionally comes the variable definitions between vertical bars, then the code block in a new line, and an `end` keyword to close the block:

```
do
  puts "Hey!"
end
```

```
do |x, y|
  x = x + y
end
```

Now that you are familiar with the concept of blocks, let's move on and take a look at how we use them.

Chapter 6 discusses block in more detail.

**Numbers** Numeric variables support three types of loops: `times`, `upto\downto`, and `step`.

`times` is available for Integer types only (Fixnum and Bignum). It executes its code block the number of times of the numeric value:

```
10.times { puts "Hey!" } # prints "Hey!" 10 times
```

```
# The next loop will print the numbers 0 to 9 on the screen
10.times do |x|
  puts x
end
```

`upto` and `downto` are also available for Integer types only. They iterate from the numeric value up to or down to the given parameter:

```
5.upto(10) { |x| puts x } # prints the numbers from 5 to 10 on the screen
```

```
# The next loop will print "Hey!" 5 times
5.downto(1) do
  puts "Hey!"
end
```

step is available to all numeric variables. It enables you to run toward a certain limit using a custom step size. The first parameter is the limit, and the second one is the step size:

```
5.step(25, 5) { |x| print "-#{x}-" }
# Output: -5-10-15-20-25-
```

```
1.step(0, -0.1) { |x| print x if x < 0.5 }
# Output: 0.4 0.3 0.2 0.1 0.0
```

**Ranges** Ranges are “native” to the task of looping. By using the range’s `each` method, you can loop over the range values:

```
(1..5).each { |x| puts x } # prints the numbers between 1 to 5
("A".."Z").each { |x| puts x } # prints the English alphabet
```

**Other Enumerable Objects** We’ve discussed here the commonly used enumerations. However, there are many more enumerable objects in the Ruby language. Actually, you will find that almost every object holds some kind of enumeration capabilities. `Array`, `Hash`, `String`, `File`, and `Dir` classes all support enumerations, and of course, there are more that do, as well:

```
[1,2,3].each { |x| print x } # prints "123"
{:a => ?a, :b => ?b}.each { |k, v| print k, v } # prints "a97b98"
Dir.new("C:\\").each { |file| puts file } # prints all files and folders in C:\
```

### Altering Loops Flow

Ruby features several keywords whose aim is to alter the execution flow of loop blocks (and blocks in general, as discussed in the next chapter). These keywords include `break`, `next`, and `redo`.

**break** When `break` is used inside a loop, the loop immediately ends, and the application flow continues to the next expression:

```
(1..1000000).each do |i|
  break if i > 5
  puts i
end
```

The preceding code prints to the screen the numbers from 1 to 5. Then, the flow reaches the `break` statement and the loop ends.

## BREAK AS AN EXPRESSION

We've discussed earlier that everything in Ruby is an expression and can return a value. `break` is no exception; it is an expression, too, and can return a value:

```
status = for elem in [ "Beagle", "Terrier", "Labrador Retriever", "Persian
➡Cat" ]
    if elem.include?("Cat") then
        break "Cat alert!"
    end
end
```

---

```
puts status # print "Cat alert!" to the screen
```

---

**next** The `next` keyword causes the current loop iteration to end. The loop then continues to the next iteration:

```
[ -2, 5, -12, 19, 43 ].each do |x|
  next if x < 0

  puts x
end
```

This code outputs to the screen the numbers 5, 19, and 43.

**redo** When the `redo` keyword is called, the current iteration starts over. Unlike the `next` keyword, which results in moving forward to the next iteration, the `redo` keyword runs the same iteration again:

```
num = 0
(1..5).each do |i|
  num += 1
  redo if num == 2
end
puts num # print 6 to the screen
```

# This code will redo the iteration in case of a wrong answer

```
1.upto(5) do |x|
  puts "-> #{x} <-"
  print "What number do you see? "
  answer = gets

  if answer.to_i != x
    puts "WRONG... Try Again:"
```

```

    redo
  end
end

puts "CONGRATULATIONS!"

```

## The `yield` Statement

You can think of `yield` as a built-in delegating mechanism. Calling `yield` executes the code block that is attached to the method call, similar to regular method invocations. After this code is done, the flow returns to the clause `yield` is declared in. Because of its nature, `yield` is commonly used in iterator implementations (each methods, for example).

You can pass parameters with `yield` and they will be passed to the executed code.

For example, in the next sample I define an enumerable method that implements the hailstone sequence. On each iteration, I use `yield` to call the associated code. The associated code takes the current value and prints it to the screen:

```

def hailstone_sequence(start, steps)
  step_index = 0
  num = start

  # Send the initial number to the associated code block
  yield num

  while (step_index < steps)
    # Calculate the next number in the sequence
    if num % 2 == 0
      num = num / 2
    else
      num = (3*num) + 1
    end

    # Send the current number to the associated code block
    yield num

    step_index += 1
  end
end

# Using the enumerable method
hailstone_sequence(100, 50) do |x|
  puts x
end

```

The code ends up printing a sequence of 50 numbers to the screen according to the hailstone sequence specifications.

### THE HAILSTONE SEQUENCE

A hailstone sequence, also known as the Collatz conjecture, is still an unsolved theory. By using the algorithm in the code, given any positive integer, the sequence eventually reaches 1. Then the sequence infinitely continues with the subsequence of 4, 2, 1.

There is \$500 waiting for whoever finds a solution to the theory. Good luck!

---

### BEGIN and END

BEGIN and END, when defined, executes their associated code block at the beginning of the application and at the end. BEGIN and END are followed by a code block that is surrounded by curly brackets:

```
BEGIN {  
    puts "Starting"  
}
```

```
END {  
    puts "Ending"  
}
```

If you run this, you'll print "Starting" and "Ending" to the screen.

These blocks can appear multiple times within the application, and they will all be executed. BEGIN clauses are executed first to last, and END clauses last to first:

```
BEGIN { puts "begin #1" }  
END { puts "end #1" }
```

```
puts "Outside BEGIN or END"
```

```
BEGIN { puts "begin #2" }  
END { puts "end #2" }
```

The preceding sample prints the following output to the screen:

```
begin #1  
begin #2  
Outside BEGIN or END  
end #2  
end #1
```



## Exception Handling

Exceptions in Ruby are similar to exceptions in other languages. They are raised when an error occurs. There are several types of exceptions in Ruby. Some add extended information about the error, but most don't. The base exception class is named `Exception`, and its derivatives are `StandardError`, `NoMethodError`, `SyntaxError`, `TypeError`, `SystemCallError`, and more.

By default, Ruby terminates the application when an error occurs. Exception handling using the `rescue` statement can prevent this from happening.

### Exception Information

As mentioned, every exception in Ruby is based on the `Exception` class. This class offers a few methods that enable you to investigate the source of the exception.

`message` retrieves a human-readable error message related to the exception:

```
begin
  # Divide by zero... this will raise an exception
  num = 5 / 0
rescue => ex
  puts ex.message
end
```

An error will be printed to the screen: Attempted to divide by zero.

`backtrace` is the stack trace of the place where the error occurred. It returns an array of the trace, each line with the filename, line number, and method name:

```
begin
  # Divide by zero... this will raise an exception
  num = 5 / 0
rescue => ex
  puts ex.backtrace
end
```

This will print the stack trace to the screen:

```
Demo.rb:3:in '/' Demo.rb:3
```

### **rescue**

The `rescue` statement is used to handle exceptions. When an exception occurs, the application flow skips all other statements and jumps right to the `rescue` statement.

`rescue` is most often used in a `begin-end` clause. When an exception occurs within the code that is written inside the `begin-end` clause, the exception gets to the `rescue` statement:

```
begin
  ... code ...
rescue
  # The flow will reach this point only in case the code between the
  # begin and rescue keywords raises an exception
  ... handling the exception ...
end
```

### RESCUE CAN BE USED WITH MORE CLAUSES

The `rescue` statement can be used with more than just the `begin-end` clause. It can also be used with method, class, and module definitions. It then catches errors raised from the scope it is written in. For example, `rescue` on a class definition will not catch errors that are raised in methods in that class.

The way to use `rescue` in a method clause is demonstrated in the following example:

```
def my_method(param)
  ... method code ...
rescue
  ... exception handling ...
End
```

---

### Receiving the Exception Object

You can also retrieve the exception object when an error occurs. With this object, you can get information about the error and handle it accordingly. To retrieve the exception, the `rescue` statement allows defining a variable that will hold the object. The name of the variable comes after an `=>` operator, which says to the `rescue` statement “put the exception there”:

```
begin
  ... code ...
rescue => ex # ex is the variable that will hold the exception once it occurs
  ... exception handling ...
  # Using the ex variable:
  puts ex.message
end
```

**RECEIVING THE EXCEPTION WHEN NO VARIABLE IS DEFINED**

If you have a rescue clause with no variable defined and you still want to access the exception object, you can do that through the special Ruby variable `$!`, as follows:

```
begin
  do_something # do_something is not defined
rescue
  print "Exception:  "
  print $!.message
end
```

This prints the following the screen:

---

```
Exception:  undefined local variable or method `do_something' for main:Object
```

---

**Handling Different Exception Types**

A rescue statement can be defined for a single exception type. This is handy when you want to handle the exceptions differently. For example, you will want to show the error to the user when a `SyntaxError` occurs and maybe retry the operation when an `IOError` occurs. This behavior is available by writing the exception class name right after the rescue statement:

```
begin
  do_something
rescue NameError
  puts "We couldn't find the method!"
rescue IOError => ioEx # can still set the exception to a variable
  puts "IO error: #{ioEx.message}"
rescue => ex # Other exceptions will end up here
  puts "Unexpected error: #{ex.message}"
end
```

**rescue as a Statement Modifier**

Similar to other statements in Ruby (like `if`), the `rescue` statement can be used as a modifier and be written on a single line. In this case, the `begin-end` clauses can be skipped:

```
x = 5/0 rescue x = 1 # x will end up set to 1
```

Because everything is an expression, the next block is identical to the preceding one:

```
x = begin
  5 / 0
rescue
  1
end
```

**retry**

The `retry` keyword, as its name declares, retries the block that has caused the `rescue` statement to start. The next code sample tries to connect to the server two times at most:

```
retries = 0
begin
  retries += 1

  contact_server
rescue
  puts "Error, trying again"
  if retries < 2
    sleep 1 # Sleep for one second
    retry
  end
end
```

The `retry` keyword proves handy for exceptions that are caused because of server time-outs, for example. It is not a good idea to use it for every exception because most exceptions inform of problems within the application flow and should be treated otherwise.

**else**

The `else` statement, as odd as it sounds, is an alternative to `rescue`. It doesn't mean that `else` catches exceptions. It means that when none of the `rescue` statements is called, the flow continues to the `else` clause.

Putting code in an `else` clause is similar to putting it at the end of the `begin` clause. The only difference is that exceptions from the code in the `else` clause will not be caught by the `rescue` statements.

```
begin
  ... code ...
rescue TypeError
  .. TypeError handling ...
rescue IOError
  ... IOError handling ...
rescue NameError
  ... NameError handling ...
rescue Exception
  ... All other exceptions handling ...
else
  # The code will reach here if none of the above rescue statements is executed
  ... code ...
end
```

This technique is quite uncommon. It might help when you want to distinguish the “rescue-protected” code and the code that should be run in case of a success.

## ensure

Code that is defined within an `ensure` clause will always run (similar to `finally` in other languages). This code will be executed in both cases when an error occurs in the `begin` clause and when the code successfully ends.

The `ensure` statement appears at the end of a `rescue` block. It is placed beneath all `rescue` statements and the `else` statement (if `else` appears). `ensure` can also be used without `rescue` statements above it. This ends up running even if the code in its `begin` clause fails.

For example, the next code fails after a file is opened. The `ensure` clause ensures that the file stream is closed:

```
begin
  # Open a file for writing
  f = File.open("demo.txt", "w")

  x = 5/0
rescue
  ... error handling ...
ensure
  # Make sure the the file stream is closed
  f.close
end
```

Remember that `else` and `ensure` have entirely different roles: `ensure` is always called, whereas `else` is called only when no error occurs. Moreover, `ensure` will be executed even after an `else` clause is run and even after a `return`, `break`, or `next`:

```
1.upto(4) do |x|
  begin
    next if x > 2

    puts x
  ensure
    puts "In ensure"
  end
end
```

What is the output of the preceding code? Think a moment about it. (Hint: It’s tricky!)

The first answer most of the people think of is this:

```
1
2
```

```
In ensure
In ensure
```

This is not the right answer, though. As previously mentioned, `ensure` always runs. This is what makes it possible to run after flow breakers like `next`. However, `ensure` will be executed also after a successful run. So the real output of the code is as follows:

```
1
In ensure
2
In ensure
In ensure
In ensure
```

## raise

Apart from catching exceptions, Ruby enables you to raise them, too. Raising exceptions proves handy when you want to indicate to the caller that something didn't go as expected.

Be aware that an exception raised manually is identical to a system-raised exception. In case no handling exists, the application terminates.

Raising an exception in Ruby is done via the `raise` statement. This statement is flexible and allows you to raise exceptions in various ways.

### Way 1: raise with No Arguments

When `raise` is used with no arguments in a `rescue` clause, it re-raises the exception:

```
begin
  5 / 0
rescue
  puts "Error!"
  raise
end
```

This code prints "Error!" to the screen and rethrows the `ZeroDivisionError` that it caught.

When `raise` is used without arguments in every clause other than `rescue`, it raises an empty `RuntimeError` exception. This technique is not recommended because the caller does not know why the error happened:

```
begin
  raise
rescue => ex
  puts ex.message # ex.message is empty, nothing will be printed out
end
```

**Way 2: raise with a String Argument**

Passing a string to `raise` throws a `RuntimeError` with the string as its message:

```
begin
  raise "Oh no!"
rescue RuntimeError => ex
  puts ex.message
end
```

The screen output is “Oh no!”

**Way 3: raise with an Exception Object Argument**

If you have generated the exception object, you can pass it to the `raise` statement. This exception will be thrown to the caller:

```
begin
  raise SyntaxError.new("There's no code here!")
rescue SyntaxError => ex
  puts "Syntax problem: #{ex.message}"
rescue
  puts "Unexpected error"
end
```

The preceding code outputs “Syntax problem: There’s no code here!” to the screen.

**Way 4: raise with an Object Argument**

This way is rather uncommon. You can pass an object to `raise`. This object should have an `exception` method that generates an exception object. The `raise` statement then invoke this method and throws the exception:

```
class SomeClass
  def exception
    RuntimeError.new("SomeClass error!")
  end
end

begin
  raise SomeClass.new
rescue RuntimeError => ex
  puts "#{ex.message}"
end
```

This outputs “SomeClass error!” to the screen.

### Way 5: rescue with Exception Details

`raise` can receive up to three arguments. The first is the exception's type, the second is the error message, and the third is the `backtrace` content. The `backtrace` argument lets you define a more human-friendly version of the stack trace.

The second and third arguments are optional here (and we've already seen that the error message can also be passed alone):

```
begin
  raise NotImplementedError, "Implementation is due next summer"
rescue NotImplementedError => ex
  puts "Not implemented [{ex.message}]"
  puts ex.backtrace
end
```

Output:

Not implemented [Implementation is due next summer]

Demo.rb:2

```
begin
  raise RuntimeError, "Code's missing", [ "demo.rb first part", "Second begin-end" ]
rescue RuntimeError => ex
  puts "Not implemented [{ex.message}]"
  puts ex.backtrace
end
```

Output:

Error: Code's missing demo.rb first part Second begin-end

This is the recommended way of raising exceptions. It gives the caller all the information it needs to handle the error correctly.

## Custom Error Classes

You can declare your own exception classes to customize them to your needs.

Creating custom exception classes is easy. All you have to do is to inherit from the `StandardError` class. (You'll read more about classes and inheritance in the next chapter.)

The most common way is to inherit from `StandardError` without adding any new functionality. The type of the error is a sufficient indicator for the error:

```
class CustomError < StandardError
end

# Or even in one line:
class CustomError < StandardError; end
```



You can, of course, add functionality to the error classes. For example, in the next custom error, `DayError`, I add the name of the related day, as well:

```
class DayError < StandardError
  attr_accessor :DayName
end
```

Using custom error classes is just like using system error classes. You can raise them and catch them in rescue clauses:

```
begin
  if (5..6).include?(Time.now.wday)
    #Construct the error object
    error = DayError.new("This is the weekend, no work!")
    error.DayName = "WEEKEND!"

    # Raise it
    raise error
  end
rescue DayError => ex
  puts "Wrong day: #{ex.DayName}, #{ex.message}"
end
```

Output: “Wrong day: WEEKEND!, This is the weekend, no work!”

## Summary

The Ruby language is a permissive and flexible language. This chapter has introduced you to its basic syntax and concepts (such as the different variable types, conditions, loops, and exception-handling techniques).

So far in this book, you’ve learned about only a small part of the Ruby language and its possibilities, which limits us to small and simple Ruby applications. The next chapters take a deeper look at the Ruby language, after which you can leverage your newfound knowledge to write larger and more complex Ruby applications.

## CHAPTER 6

# Ruby's Code-Containing Structures

### IN THIS CHAPTER

- ▶ Methods
- ▶ Blocks, Procs, and Lambdas
- ▶ Classes
- ▶ Modules

With Ruby's basic syntax only, we still find it hard to build a full Ruby application.

For example, sometimes when building an application you need to write the same code several times. If you really write it several times, you end up shortly with an impossible-to-maintain and buggy application. The solution to that is to write the code once in a central place and refer to it from several places.

Another common example is code design. One important task in building big applications is designing your code to make it more maintainable, efficient, and change-ready. We can't implement such design decisions currently because we don't yet know how to separate our code to logical parts.

In this chapter, you learn how to organize the building blocks discussed in the preceding chapter into bigger units that can help you write code in a more efficient, easier-to-maintain, and logical way.

## Methods

Methods are code blocks that have names and parameters. In Ruby, the last executed expression is returned from the method to its caller. Therefore, there are no methods in Ruby that do not return a value (sometimes called procedures). If the last expression evaluated does not consist of a value, `nil` is returned.

Methods can be associated with an object. (In most cases, the object will be a class.) When a method is associated with an object, the method is executed within the object

context. Previous code samples have shown methods that are not associated with any object. This is absolutely acceptable in Ruby. However, because Ruby is an object-oriented language, the “no-object” methods are actually implicitly defined and invoked as private members of the `Object` class.

For information about calling methods, see Chapter 5, “Ruby Basics.”

## Defining Methods

Methods are defined using the `def` keyword. The `def` keyword is followed by the method name and parameters list if needed, optionally surrounded with parentheses. After the list of parameters, the method code appears, and then the `end` keyword finishes the method definition:

```
def add_one(num)
  puts num + 1
end
```

This method is named `add_one`. It receives one argument called `num`. The method then adds 1 to the received number and prints it to the screen. Because `puts` returns `nil`, and it's the last (and first) expression invoked during the method execution, the method returns `nil` to the caller, as well.

Methods do not have to contain arguments. In case a method does not have parameters, only the method name follows the `def` keyword:

```
def say_hello
  puts "Hello!"
end
```

## CATCHING EXCEPTIONS WITHIN METHODS

When you define a method, `def` acts as the `begin` clause that is needed for the `rescue` statement. Therefore, you don't have to write the following:

```
def my_method
  begin
    ...code...
  rescue
    ... handle error...
  else
    ...code...
  ensure
    ...code...
  end
```

The following code also works (and is more readable and recommended):

```
def my_method
  ...code...
rescue
  ... handle error...
else
  ...code...
ensure
  ...code...
end
```

---

## MULTIPLE-PARAMETER REPLACEMENT

It is common in Ruby to “squeeze” multiple logically related attributes into a single parameter and pass it as a hash.

For example, take a look at the next code sample:

```
def print_person(first_name, last_name, age, country)
  puts "#{first_name} #{last_name}"
  puts "#{age} years old, from #{country}"
end
```

```
print_person("John", "Doe", 25, "USA")
```

You should change it to the following code block, which other developers will find more readable and easier to use:

```
def print_person(details)
  puts "#{details[:first_name]} #{details[:last_name]}"
  puts "#{details[:age]} years old, from #{details[:country]}"
end
```

```
print_person(:first_name => "John",
             :last_name => "Doe",
             :age => 25,
             :country => "USA")
```

Another common use for hash parameters is for optional parameters. The required ones are direct parameters, and the optional ones are given via the hash.

---

## Method Naming

As discussed in Chapter 5, methods are named using lowercase letters and an underscore to separate the words.

Two important naming conventions are used widely across the language. They are recommended to make the code more readable.

Methods that are expected to return a Boolean value should have a question mark at the end of their name: `nil?`, `empty?`, `include?`.

Methods that should be carefully used, or that change the internal data of the object, should have an exclamation mark at the end of their name: `sort!`, `strip!`, `merge!`. The difference, for example, between `String.reverse` and `String.reverse!` is that `reverse` returns a new reversed `String` object and `reverse!` changes the same string:

```
str = "Hello"
another_str = str.reverse # str = "Hello", another_str = "olleH"
str.reverse! # str = "olleH"
```

## Returning a Value from Methods

Methods in Ruby always return a single value. This value can be of any object: numeric, string, array, or any other object. The sole case when a method will not return a value is in case of an exception.

Generally, the result of the last executed line is the return value of a method:

```
def get_hour_name(hour)
  case hour
  when 0, 24
    "Midnight"
  when 1..11
    "Morning"
  when 12
    "Noon"
  when 13..20
    "Afternoon"
  when 21..23
    "Night"
  end
end
```

This method returns the name of the part of day for a given hour. If the hour is 0 or 24, “Midnight” is returned. For values between 1 and 11, “Morning” is returned, and so on.

Another way to return a value from a method is by using the `return` keyword. The `return` keyword stops the method execution. If an expression follows it, the expression is evaluated, and its result is returned to the method caller.

The `return` keyword can be also used to explicitly return a value. For example, there is no difference between the next two method definitions:

```
def add_one(num)
  num + 1
end
```

```
def add_one(num)
  return num + 1
end
```

Another important feature the `return` keyword offers is returning multiple values. You can pass the `return` keyword multiple values separated by commas, and they will be sent to the caller as an array:

```
def get_country_details(country_name)
  if country_name == "USA"
    return "USA", 1776
  elsif country_name == "IL"
    return "Israel", 1948
  else
    return "unknown", 0
  end
end
```

This return value can be set into different variables directly:

```
name, declared_on = get_country_details("USA")
```

## Method Name Aliasing

In many places in Ruby, you can find multiple methods that do exactly the same task. This is an ingenious approach. It increases the readability of the code and makes it clearer to the developers, who can use the term that makes more sense to them.

Creating an alias name for a method is done with the `alias` keyword. The way to define an alias is by passing `alias` the alias name as the first argument and the method name as the second one:

```
def merge
  ...code...
end

alias unite merge
```

Now you can call `unite` rather than `merge`.

Another common use-case for method aliasing is to replace a method in a class and then still be able to call the old method. For example, the next code replaces the `the_next_big_thing` method but keeps the original implementation using method aliasing:

```
def the_next_big_thing
  puts "Other languages"
end
alias the_previous_big_thing the_next_big_thing
def the_next_big_thing
  puts "IronRuby"
end
the_next_big_thing # Prints "IronRuby"
the_previous_big_thing # Prints "Other languages"
```

## Default Parameter Values

Method arguments can have default values. These values are used when no parameters are passed in their position.

The way to declare a default value is by setting it to the parameter when it is defined. For example, in the next sample, the `size` argument has a default value of 1:

```
def get_substring(str, size = 1)
  str[0..size-1]
end

get_substring("Hello", 3) # = "Hel"
get_substring("Hello") # = "H"
```

The default value is evaluated every time it is needed (not once in a method lifetime). This behavior makes it available to set a default value according to other values. For instance, the next sample redefines the `get_substring` method so that the whole string is returned if `size` is not defined:

```
def get_substring(str, size = str.size)
  str[0..size-1]
end

get_substring("Hello") # = "Hello"
```

## POSITION OF ARGUMENTS WITH DEFAULT VALUES

Parameters with default values should always be positioned after parameters without default values, if they exist. Otherwise an error occurs:

```
def method(a = 1, b, c, d = 5) # Syntax error!
  def method(b, c, a = 1, d = 5) # OK
```

## Special Parameter Types

Ruby methods have two special parameter types. These are the `*` parameters and the `&` parameters. The `*` parameter, also known as the array parameter, is a “catch all parameters” argument that can be used when the number of parameters is unknown. The `&` parameter, also known as a block argument, allows passing a code block to the method.

### The Array Parameter (`*`)

We have already met the `*` operator in Chapter 5, where it helped us flatten arrays. Here its role differs just a bit. The `*` operator is used to define a parameter that catches an unknown number of arguments passed to the method. The array parameter contains an array of the caught values. If no values are “caught,” the argument contains an empty array.

This special parameter type can appear along with regular parameters, and in that case, it receives the rest of the passed arguments:

```
def concat(*values)
  total = ""
  values.each { |str| total = total + str }
  total
end

concat("Iron","Ruby") # values = ["Iron", "Ruby"]
concat() # values = []

def concat(first, *values)
  values.each { |str| first = first + str }
  first
end

concat("Iron","Ruby", " Unleashed") # values = ["Ruby", " Unleashed"]
concat("IronRuby") # values = []
```

The array parameter must be positioned after regular parameters. The only argument that may appear after it is the block argument.

### The Block Argument (`&`)

The block argument is used to send code blocks to the method. These blocks can then be invoked during the method execution.

This approach is similar to the `yield` keyword discussed in Chapter 5. The difference is that the block argument provides more control over the code block. You can investigate it (for example, discover how many arguments it expects), and you can pass it on to other methods.

Similar to the array parameter, to define a block argument, an ampersand, `&`, should be added before the parameter name. A block argument is actually a `Proc` object, and so unlike `yield`, invoking it is done via the `call` method:



```
def concat(first, *values, &result_processor)
  # Calculate the return value
  values.each { |str| first = first + str }
  # Invoke the block attached to the method call
  result_processor.call(first)
  # Return a value to the caller
  first
end
```

Calling a method with a block argument is the same as passing a code block to a method with a `yield` keyword:

```
concat("Iron", "Ruby") { |result| puts result }
```

This prints “IronRuby” to the screen.

### BLOCK ARGUMENT POSITION

A block argument must be the last parameter in the list. Otherwise an error will be raised.

---

### USING METHODS AS BLOCK ARGUMENTS

Instead of passing a new code block, you can pass the method every object that can be converted to a Proc object. This includes other methods. For example, we could write the `concat` method call this way:

```
def print_string(str)
  puts str
end

concat("Iron", "Ruby", &method(:print_string))
```

The method named `method` is actually an Object class method, and it returns the Method object of a given method. The ampersand at front actually converts the Method object to a Proc object. Eventually, the method is executed, and we get the same result as before.

---

## Associate Methods with Objects

You can associate methods to every object out there, even to constants. This mechanism is also called *singleton methods* because the method is available only for a single object.

To define a singleton method, the `def` statement, instead of writing it like we’ve already seen (`def [method name]`), is written along with the associated object name, `def [object name].[method name]`, as follows:

```
str = "Hello"
def str.welcome?
  self == "Hello"
end

str.welcome? # = true
```

The `welcome?` method is available only for the `str` object. It is not available to any other string, clone of `str`, or object. The method will be accessible during the whole lifespan of `str` and no longer.

There is another syntax for creating singleton methods. It consists of a class-like definition. The equivalent code of the `welcome?` sample using this syntax is as follows:

```
str = "Hello"
class << str
  def welcome?
    self == "Hello"
  end
end

str.welcome?
```

This syntax is a bit more complex. First you must declare that you are modifying the class of `str` (`class << str`), and then you can write the singleton method definition.

## OBJECTS YOU CANNOT ASSOCIATE METHODS WITH

Because numeric values (particularly `Fixnum`) and symbol values are not object references in Ruby, you cannot associate methods with their objects. You can, however, associate methods with the classes themselves (`Fixnum`, `Symbol`).

## Removing Method Definitions

Ruby allows methods to be undefined. When this is done, the method is no longer accessible. The way to return the method is to define it again.

You can remove a method definition via the `undef` keyword. It receives the method name and removes its definition:

```
def add_one(num)
  num + 1
end

add_one(3) # works
undef add_one
add_one(3) # undefined method error!
```

Another way to remove method definitions is by using the `undef_method` method. The difference between `undef_method` and `undef` is that `undef_method` is a private member of `Module` and so it can be used only within classes or modules. In addition, `undef_method` receives a symbol representing the method name and not the method itself.

## UNDEF AND SINGLETON METHODS

As mentioned earlier, singleton methods can be associated with any object. This kind of method *cannot* be removed using the `undef` keyword.

To remove singleton methods, you should use `undef_method` with the class instance extension syntax. For example, if you want to remove the `welcome?` method from `str`, you should write the following:

```
class << str
  undef_method(:welcome?)
end
```

---

## Blocks, Procs, and Lambdas

Methods are code containers. They have a name to uniquely identify them, and they contain code inside. In Ruby, methods are not the sole code containers. Ruby features additional ways to pass around code blocks (for example, via blocks, procs, and lambdas). They are similar to methods, just without the name; they are anonymous. These code blocks are widely used throughout the Ruby language because they're easy to use and intuitive.

### Blocks

Blocks in Ruby are anonymous methods. Which means that they are similar to regular methods but with a main difference: They do not have a name. This characteristic enforces the biggest limitation of blocks, which is they must be declared along with a method call. In addition, blocks are not objects and cannot be treated as such. You can't pass them around and you cannot execute operations on them. You can only invoke them with or without parameters.

Blocks can be defined in two ways: the curly brackets way and the `do-end` way. The variables the block expects are declared between vertical bars (`|`) after the beginning of the block clause:

```
# Curly brackets block:
some_method_call { |x| puts x }
# Multiline also:
some_method_call { |x|
  puts x
}
```

```
# begin-end block:
some_method_call do |x| puts x end
# Multiline also:
some_method_call do |x|
  puts x
end
```

Blocks were discussed in Chapter 5 in relation to iterators. However, iterators are only one possible implementation. In fact, a block can be added to any method. It is possible even if the method does not have code that invokes it (`yield` or an ampersand parameter). In this case, the block will be quietly ignored.

## BLOCKS ARE CLOSURES

Blocks in Ruby are closures. This means that they carry their context with them.

For example, in the following code, the block can access the `num` variable even though it is not in the iteration context:

```
num = 1
[1,2,3].each { |x| print x+num } # Prints 234 to the screen
```

---

## Procs

Procs enable you to save blocks into variables. This gives them the ability to be passed around and reused. Procs consist of the `Proc` class that contains methods that provide information about the proc.

Defining a `Proc` object is done by using the `Proc` class constructor, `Proc.new`, as follows:

```
# Using curly brackets:
p = Proc.new { |x| puts x }
# Using do-end:
p = Proc.new do |x| puts x end
```

Because a `proc` is an object, it cannot be invoked directly like methods. The alternative to that is using `Proc`'s `call` method. The parameters targeted to the block are passed to the `call` method:

```
p.call("Hello") # Prints "Hello" to the screen
```

Another way to execute a `proc` is by calling it with square brackets:

```
p["Hello"] # identical to p.call("Hello")
```

This way is more similar to method invocation. Unlike regular methods, when the block doesn't receive any arguments, square brackets will still be needed:

```
p = Proc.new { puts "Hello" }
p[] # prints "Hello" to the screen
```

### Proc Class Methods

The Proc class has several methods that allow getting information about the code block.

`arity` returns the number of parameters the block expects:

```
p = Proc.new { |x| }
p.arity # = 1

p = Proc.new { |x, y| }
p.arity # = 2
```

There are some behaviors to notice regarding the `arity` method. For blocks with no parameters, it returns `-1`:

```
p = Proc.new { }
p.arity # = -1
```

After the block is defined to receive an asterisk parameter, `arity` returns the number of arguments as a negative number (including the asterisk parameter):

```
p = Proc.new { |*x| }
p.arity # = -1

p = Proc.new { |x, y, z, *t| }
p.arity # = -4
```

`to_s` generates a string representation of the Proc object. This string contains some useful information, such as the file and line number where the object was defined. Its format is as follows:

```
p = Proc.new { puts "Hello" }
p.to_s # = #<Proc:0x03c6fe3c2>
```

Figure 6.1 shows the parts of Proc's `to_s` output.

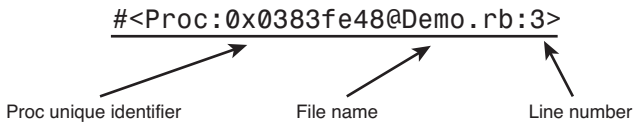


FIGURE 6.1 The different parts of the result of the Proc's `to_s` method.

## Lambdas

Lambdas are similar to procs. Actually, lambdas and procs both consist of the Proc class. Calling them is done the same way. They both have Proc methods available (like `arity`), and their syntax is very close. They differ in their behavior. Procs have a block-like behavior, and lambdas have a method-like behavior.

### Defining Lambdas

A lambda is defined using the `lambda` keyword. Just like blocks, after the `lambda` keyword can appear a curly-brackets block or a `do-end` block:

```
l = lambda { |x| puts x }
l = lambda do |x| puts x end
```

### Differences Between Lambdas and Procs

The differences between lambdas and procs find expression on a few occasions. One occasion is when too few arguments are passed to the block. A proc sets the missing parameters to `nil`, whereas a lambda raises an error:

```
p = Proc.new { |x, y| print x; print y }
l = lambda { |x, y| print x; print y }

p.call(1) # Prints "1nil"
l.call(1) # Error!
```

Another case is when the `return` keyword is used inside the block. Lambdas, just like methods, return this value to the invoking method. For example, the next code sample ends up printing "In Lambda, result = 1":

```
l = lambda { print "In Lambda"; return 1 }
result = l.call
print ", result = #{result}"
```

The behavior of procs can be slightly unexpected. When defined and invoked within the same context, a `return` statement inside a proc will be treated just like it happened within the invoker context. For instance, the following code sample ends up printing only "In Proc":

```
def my_method
  p = Proc.new { puts "In Proc"; return 1 }
  p.call
  puts "Outside"
end
```

```
result = my_method # result will contain 1 after the method invocation
```

When the proc is not defined and executed within the same context, an error occurs:

```
def my_method(p)
  p.call
  puts "Outside"
end
```

```
my_method( Proc.new { puts "In Proc"; return 1 } ) # LocalJumpError!
```

## Flow-Altering Keywords Within Blocks, Procs, and Lambdas

The keywords we discuss in this section have already been mentioned in the context of loops (in Chapter 5). Loop blocks are like every other block in Ruby, and so it is natural that what you can do in one block can be done in another. This is correct about blocks, but not entirely right about procs and lambdas.

`next` is available in blocks, procs, and lambdas. It is used to stop the block execution and continue the application flow. When it appears with a value afterward, this value is returned to the invoker:

```
def demo(m1, m2)
  block_result = yield
  proc_result = m2.call
  lambda_result = m1.call
  print "#{block_result}, #{proc_result}, #{lambda_result}"
end
```

```
p = Proc.new { next "From Proc"; puts "Hello" }
l = lambda { next "From Lambda"; puts "Hello" }
```

```
demo(l, p) { next "From Block"; puts "Hello" }
```

This code will print “From Block, From Proc, From Lambda” to the screen.

redo is also allowed within all block types. When used, it restarts the block execution:

```
p = Proc.new do
  puts "IronRuby rocks? [Y|N]"
  answer = gets.chomp # Get the answer without the enter char at the end
  redo if answer != "Y"
end

p.call
```

This code asks the user “IronRuby rocks?” and waits for the user to insert input. If the input is other than Y, the question is asked again.

break behaves identically to the return keyword mentioned earlier in this chapter. It is available within blocks and within procs from the same context. break is not available in lambdas at all.

## Classes

Ruby is a purely object-oriented language. Everything is an object or acts like one. Every object consists of a class. For instance, we have mentioned that even the Boolean values, true and false, are actually TrueClass and FalseClass.

Classes are containers. They contain methods, attributes, and variables, which all serve the purpose of the class. Conforming to the OOP guidelines, classes in Ruby can inherit and extend their superclasses. Ruby classes also have control over the visibility of their methods, defining them as public, protected, or private.

Although classes in many languages are very strict, classes in Ruby are very permissive. You can “open” every class and add methods to it, and you can even add methods to a single object instance, as discussed in relation to singleton methods.

## Defining Classes

The class keyword is used to define a class. The keyword is followed by the class name. The class definition is ended with the end keyword. Class names in Ruby *must* start with a capital letter. An error will be raised otherwise:

```
class ClassName
  ...class content...
end
```

Class names must start with a capital letter because Ruby creates a constant with the same name to refer to the class.



## Creating a Class Instance

To use a class, we should instantiate it first. The way to do that is by calling the `new` method, which is the constructor of every class in Ruby:

```
human = Human.new
```

Every class has two methods that can help identify the class of the current variable instance. These methods are `class` and `is_a?`:

```
human.class # gets the class name of the variable = "Human"
human.is_a?(Human) # tests if a variable consists of the class Human = true
```

## Defining a Constructor

When the `new` method is called, an instance of the class is created, and the class constructor is executed. The name of the constructor method in Ruby is `initialize`. You can add parameters to the method, and they will be passed along by the `new` method:

```
class Human
  def initialize(first_name, last_name)
    print "#{first_name} #{last_name}"
  end
end

Human.new("John", "Doe") # prints "John Doe"
```

### INITIALIZE VISIBILITY

The `initialize` method is used as the constructor method. Ruby automatically makes `initialize` a private method, so you can't call it from outside the class:

```
human = Human.new
human.initialize # Error!
```

This also means that calling the `initialize` method is available from instance methods. Calling it from there, although doing so makes no sense, will not harm the application flow or create a new class instance. `initialize` is just a regular method with a special role that is automatically called *after* the class instance has been created.

## Variables Inside Classes

Talking about classes is a bit different from talking about global objects. Classes have instances, and their inside state differs from one instance to another. This is why classes can contain more than one variable type. Actually, a class has four different variable types: class variables, instance variables, local variables, and constants.

### Class Variables

A class variable is similar to static variables in other languages. A class variable is single for every class and is not affected by different instances of the class. Therefore, if a new value is set to a class variable, this new value is visible to all instances of this class.

To define a class variable, add two “at” signs (@) before its name (for example, @@class\_variable\_name).

The next code sample uses a class variable @@instances\_count to count the number of instances of the Demo class:

```
class Demo
  @@instances_count = 0

  def initialize
    @@instances_count += 1
  end

  def print_instances_count
    puts @@instances_count
  end
end

c = Demo.new
c.print_instances_count # prints 1

d = Demo.new
d.print_instances_count # prints 2
```

### Instance Variables

An instance variable exists within a single class instance. Unlike class variables, changing an instance variable on one instance of a class will not affect its value on other instances. Instance variables are very useful in saving data that is needed across the class methods. It is a common use to set instance variables with data provided on the initialize method.

Defining an instance variable is done by adding a single “at” sign (@) before the variable name (for example, @instance\_variable\_name).

In the next code sample, we improve the Human class and make it save the data passed to the constructor. In the introduce method, we use the data and print it out:

```
class Human
  def initialize(first_name, last_name)
    @first_name = first_name
    @last_name = last_name
  end
```

```

def introduce
  puts "Hi, I'm #{@first_name} #{@last_name}"
end

john = Human.new("John", "Doe")
joanne = Human.new("Joanne", "Doe")

john.introduce # prints "Hi, I'm John Doe"
joanne.introduce # prints "Hi, I'm Joanne Doe"

```

### USING INSTANCE VARIABLES OUTSIDE INSTANCE METHODS

Coming from languages like C# or Java, you might find it logical to initialize instance variables in this way:

```

class Demo
  @number = 0

  def demo_method
    # Use the number instance variable
    @number += 1
    puts @number
  end
end

```

---

This is wrong in Ruby and will end up with an error when trying to execute `demo_method`.

Code that isn't defined within instance method clauses relates to the class object itself. Like class variables, the `@number` variable we have defined relates to the class and not to its instances. This is why it cannot be accessed through instance-related code. It will be accessible to class methods, as discussed later in this chapter.

To write the preceding sample correctly, we just need to move the variable initialization to the `initialize` method:

```

class Demo
  def initialize
    @number = 0
  end

  def demo_method
    # Use the number instance variable

```

```

    @number += 1
    puts @number
end
end

```

### Local Variables

Local variables are the same method variables we've already seen. A variable, which is defined inside a method, will "live" during a single method invocation only. It will not be accessible from outside the method or from the same method on a different invocation:

```

class Demo
  def initialize
    @first_time = true
  end

  def method1
    # Add 1 to num only if this method was called before
    if (!@first_time)
      num += 1 # Error! num still hasn't been initialized!
    end

    num = 0
    @first_time = false
  end

  def method2
    num += 1 # Error! num doesn't exist in this scope
  end
end

```

### Constants

Constants, as mentioned in Chapter 5, are similar to variables. Inside classes, constants are similar to class variables. They are class object variables and not instance related.

The way to define class constants is no different from regular constants:

```

class Human
  NUMBER_OF_LEGS = 2

  def initialize(first_name, last_name)
    @first_name = first_name
    @last_name = last_name
  end
end

```

```

def introduce
  puts "Hi, I'm #{@first_name} #{@last_name}"
  puts "I have #{NUMBER_OF_LEGS} legs"
end
end

Human.new("John", "Doe").introduce # = "Hi, I'm John Doe
                                   #   I have 2 legs"

```

Constants differ from the other variable types in their accessibility. They are, by default, accessible from outside the class. From within the class, constants can be accessed by their name. From outside, they should be called with the name of the class and two colons before their name:

```
Human::NUMBER_OF_LEGS # = 2
```

### Accessing Variables from Outside

Class, instance, and local variables cannot be accessed from outside the class:

```

class Demo
  @@var1 = "I'm var #1"

  def initialize
    @var2 = "I'm var #2"
    var3 = "I'm var 3"
  end
end

c = Demo.new
c.@@var1 # Error!
c.@var2 # Error!
c.var3 # Error!

# Accessing the variables from the class object won't work either:
Demo::@@var1 # Error!
Demo::@var2 # Error!
Demo::var3 # Error!

```

To make these variables available from outside, we have to add getter methods that acquire them:

```

class Demo
  @@var1 = "I'm var #1"

  def initialize
    @var2 = "I'm var #2"

```

```

    var3 = "I'm var 3"
end

def get_var1
  @@var1
end

def get_var2
  @var2
end

def get_var3
  # var3 is a local variable of the initialize method, it doesn't exist in this
  scope!
end
end

c = Demo.new
c.get_var1 # = "I'm var #1"
c.get_var2 # = "I'm var #2"

```

This approach is okay, but it's annoying to write this code again and again. Ruby has a much simpler solution: accessors.

## Accessors

Accessors are the Ruby way to automate the writing of getter and setter methods. A variable that has been created using accessors is called an *attribute*. There are four types of accessors in Ruby: `attr`, `attr_reader`, `attr_writer`, and `attr_accessor`. They differ in access level. `attr_reader` creates a read-only attribute (getter), `attr_writer` generates a write-only attribute (setter), and `attr_accessor` produces a read-write attribute. `attr` creates a getter attribute, but it receives another parameter that makes it possible to tell it to add a setter also.

To use these accessors, the attribute name should be passed to them. The convention is to use symbols as the attribute names (Ruby also accepts strings):

```

class Demo
  attr_reader :read_only_attribute
  attr_writer :write_only_attribute
  attr_accessor :read_write_attribute
  attr :read_only_attribute2
  attr :read_write_attribute2, true
end

```

Except for the `attr` accessor, the rest support receiving multiple attribute names on the same line:

```
class Demo
  attr_accessor :att1, :att2, :att3
end
```

Calling the accessors is done by the symbol name (no need for the colon at start). If you want to set a value to a read-only attribute, you can do so only within instance methods. To do that, you need to access an instance variable, which is added automatically “behind the scenes” to store the attribute data. It is named after the attribute name. So, if your attribute is named `:my_attribute`, an instance variable named `@my_attribute` is added automatically to the class. An alternative for that is calling the attribute via the `self` object, `self.my_attribute`:

```
class Human
  attr_reader :last_name
  attr_accessor :first_name

  def initialize(first_name, last_name)
    # Set the constructor params to the attributes
    @first_name = first_name
    @last_name = last_name
  end

  def introduce
    puts "Hi, I'm #{self.first_name} #{self.last_name}"
  end
end

h = Human.new("John", "Doe")
h.first_name # = "John"
# Set a new value to the first_name attribute
h.first_name = "Mo"

h.last_name # = "Doe"
h.last_name = "Shmo" # Error! This is a read-only attribute!

h.introduce # = "Hi, I'm Mo Doe"
```

## ACCESSORS “UNDER THE HOOD”

Accessors are actually methods. They belong to the `Module` method, which is the superclass of the `Class` method. The `attr_reader`, `attr_writer`, `attr_accessor`, and `attr` do some metaprogramming behind the scenes and add code to your class.

When you declare a read-write attribute

```
attr_accessor :first_name
```

this line, in fact, translates to

```
# Getter method
```

```
def first_name
```

```
  @first_name
```

```
end
```

```
# Setter method
```

```
def first_name=(value)
```

```
  @first_name = value
```

```
end
```

---

## Methods

Methods, which are defined within classes, like variables, can be instance related or class related (static in other languages). Instance-related methods are called *instance methods* and class object related methods are called *class methods*.

### Instance Methods

Instance methods are the common methods written in classes. They are associated with class instances and can access the current instance variables.

Defining an instance method is the same as regular methods. The one thing to make sure is that the method is placed within a class scope:

```
class Demo
  def instance_method(param)
    ... method code ...
    ... can define and access @ variables here...
  end
end
```

### Class Methods

Class methods, known as static methods in languages such as C# or Java, are methods that are associated with the class object itself and not its instances.

There are a few ways to define a class method. The most common one is to add the class name before the method name. In the following example, I declare a class method named `class_method`:



```
class Demo
  def Demo.class_method
    puts "Inside a class method"
  end
end
```

Another way to define a class method is by using `self` before the method name. This makes the code a bit less clear, but in case the class name is changed, you will not need to also change the class method names:

```
class Demo
  def self.class_method
    puts "Inside a class method"
  end
end
```

The third way to define class methods is by creating a special scope for class methods. This way is good if you want to make a sharp separation between instance and class methods. There are two ways to do that, inside the class definition and outside the class definition:

```
# Inside the class definition
class Demo
  ...instance methods...

  # Class methods scope
  class << self
    def class_method
      puts "Inside a class method"
    end
  end
end

# Outside the class definition
class Demo
  ...instance methods...
end

# Class methods scope
class << Demo
  def class_method
    puts "Inside a class method"
  end
end
```

The association with the class object forces a restriction upon class methods—instance variables cannot be accessed from within them. This restriction applies to class attributes, too (because they reference an instance variable):

```
class Demo
  @@class_var = 1

  def initialize
    @instance_var = 1
  end

  def Demo.class_method
    @instance_var += 1 # Error! @instance_var isn't defined within the class context!
    @@class_var += 1 # OK!
  end
end
```

An exception is instance variables that have been defined within the class context (and not the instance scope). These variables, although they are written like instance variables (with one “at” sign), operate just like class variables, and using them this way should be avoided.

Calling class methods is done without the need of a class instance. These methods relate to the class itself, so calling them is done via the class object:

```
class Demo
  def Demo.class_method
    puts "Inside a class method"
  end
end

Demo.class_method # prints "Inside a class method"

d = Demo.new
d.class_method # Error! class_method does not exist on the instance context
```

## Operator Overloading

Like other programming languages, Ruby can define new behavior to different operators.

Table 6.1 describes the operators available for overloading. The Number of Parameters column lists the number of arguments the operator method should accept.

To define the behavior of these operators, we define methods with their names and their requested number of arguments.

TABLE 6.1 Available Operators for Overloading

| Operator                           | Number of Parameters | Description   |
|------------------------------------|----------------------|---|
| <code>+ - * / % **</code>          | 1                    | Arithmetic operators (addition, subtraction, multiplication, division, modulo, and exponentiation). |
| <code>+@ -@</code>                 | 0                    | Unary plus and unary minus.   |
| <code>&lt;&lt; &gt;&gt;</code>     | 1                    | Shift-left and shift-right.   |
| <code>&amp;   ^ ~</code>           | 1                    | Bitwise-and, bitwise-or, bitwise-xor, and bitwise-complement.                                       |
| <code>&lt; &lt;= =&gt; &gt;</code> | 1                    | Order comparison.   |
| <code>==</code>                    | 1                    | Equality.   |
| <code>===</code>                   | 1                    | Case comparison.  |
| <code>=~</code>                    | 1                    | Pattern matching.   |
| <code>&lt;=&gt;</code>             | 1                    | Comparison.   |
| <code>[]</code>                    | N                    | Array access. Can retrieve any number of parameters.  |
| <code>[]=</code>                   | N                    | Array access setter. Can retrieve any number of parameters.   |

To define the behavior of these operators, we define methods with their names and their requested number of arguments.

### RETURN VALUE OF OPERATOR METHODS

It is a common technique to return the current object or a new object of the same class from the operator method. Users expect, for example, that using the plus operator returns the same object type. This expectation applies to almost all operators. Redefining these operators to return a different type might make the code hard to understand and to maintain.

### Arithmetic Operators (`+` `-` `*` `/` `%` `**`)

Arithmetic operators are usually used to perform numeric operations. However, they are not restricted to this role only. Every class that can take advantage of these operators might do so. For example, the `String` class defines the `+` operator to concatenate two strings.

In the next code sample, I add a plus operator to the `Human` class, which allows a human to have a baby (no minus operator for that!):

```
class Human
  def +(human)
    # Baby's name will be a combination of the
    # first 2 characters from the parent names
    Human.new(self.first_name[0..1] + human.first_name[0..1], self.last_name)
```

```

    end
end

john = Human.new("John", "Doe")
joanne = Human.new("Joanne", "Doe")

baby = john + joanne
baby.introduce # prints "Hi, I'm JoJo Doe"

```

### Unary + and - Operators

The unary operators are used on numeric objects to indicate whether the number is positive or negative. By default, the unary plus operator is not needed, but it is available for redefining.

I'll add a new attribute to the Human class: `asleep`. It will indicate whether a human is asleep or awake. Next I'll define the unary minus operator to indicate that the human is sleeping:

```

class Human
  attr_accessor :asleep

  def -
    # Set state
    self.asleep = true
    # Return self for ease of use
    self
  end
end

john = Human.new("John", "Doe")
john.asleep # = nil (will be treated as false on boolean expressions)
# Use the unary minus operator
john = -john
john.asleep # = true

```

### Shifting Operators

The `<<` and `>>` operators are used as shifting operators in Ruby. For numeric objects, they are used to shift the bits of a number. The shift-left (`<<`) operator is also used as the append operator. The Array class, for instance, allows adding new items to the array by using this operator.

### The Case Comparison Operator

The case comparison operator (`===`) is not commonly used. It is mostly used implicitly by the case statement. This operator doesn't necessarily compare two objects. For example, the Range class defines the `===` operator to check for membership within the range.

I'll add the case operator to the Human class and define it to check whether the parameter equals the first or last name of the current instance:

```

class Human
  def ==(other)
    if other.first_name == self.first_name || other.last_name == self.last_name
      true
    else
      false
    end
  end
end

john = Human.new("John", "Doe")
joanne = Human.new("Joanne", "Doe")
shay = Human.new("Shay", "Friedman")
case john
  when shay
    puts "Shay and John are similar"
  when joanne
    puts "Joanne and John are similar"
end
# This will print "Joanne and John are similar"

```

### The Comparison Operator

The comparison operator (`<=>`) is used to compare two objects. Pay attention that the standard return value of the comparison operator is `1`, `0`, `-1`, and `nil`. `1` stands for left is greater than right, `0` means both sides are equal, `-1` stands for left is less than right, and `nil` means the sides cannot be compared.

### The Array Access Operator

The array access operator (`[]`) can be redefined, too. This operator can receive multiple parameters, just like accessing a multidimensional array.

I will add the array access operator to the `Human` class. It will allow to access part of the human's full name. To do that, I request two parameters, `start_index` and `end_index`:

```

class Human
  def [](start_index, end_index)
    "#{self.first_name} #{self.last_name}"[start_index..end_index]
  end
end

john = Human.new("John", "Doe")
john[2, 6] # = "hn Do"

```

### The Array Access Setter Operator

The array access setter operator (`[]=`) is used to assign values to a specific array index. It can, like the array access operator, receive multiple variables.

I will add this operator to the Human class, allowing its users to change a part of the full name:

```
class Human
  def []=(start_index, end_index, new_value)
    # Set the combination of the first and sure names to a single variable
    full_name = "#{self.first_name} #{self.last_name}"

    # Update the full name string
    full_name[start_index..end_index] = new_value

    # Set the new first and last name after
    # the change (done by splitting the full name by a space char)
    @first_name, @last_name = full_name.split(" ")
  end
end

john = Human.new("John", "Doe")
john[1,5] = "ack B"
john.introduce # = "Hi, I'm Jack Boe"
```

It is important to notice that once you define the []= operator method to retrieve multiple arguments, calling the operator will be done by sending the first parameters inside square brackets and the last parameter after the equal sign.

## Special Methods

A few methods in Ruby have special uses. Some give new capabilities to the class by redefining them, and some help to handle invocation errors.

### to\_s

The to\_s method generates a string representation of the current object. This is the default method that's called when Ruby needs to implicitly convert an object to a string. Most classes in Ruby redefine to\_s to give more details than the default implementation does. For example, the Array class joins all of the array members to one string.

Our Human class doesn't redefine the to\_s method. This is the reason for the output of the next code block:

```
human = Human.new("John", "Doe")
puts human # prints "#<Human:0x381b250>"
```

The output is the name of the class and the memory address. Let's redefine to\_s to return the full name of the current human instance. Note that this method returns a string and doesn't print it out by itself:

```
class Human
  def to_s
    "Human: #{self.first_name} #{self.last_name}"
  end
end
```

```
human = Human.new("John", "Doe")
puts human # prints "Human: John Doe"
```

**each**

`each` is used to give a class the capability to iterate through its inner objects. It has no default definition. If a class has a few ways to iterate through, it is recommended to create multiple `each` methods. For example, the `String` class defines `each`, `each_line`, and `each_byte` iterator methods.

**succ**

The `succ` method role is to return the successor of the current object. For example, for `Fixnum`, `succ` returns the current number plus one. This method is necessary if you want to make your class to work with ranges.

To have a “rangeable class,” you must also provide an implementation to the `<=>` operator (which is what the range uses internally to know when it should stop).

I’ll add these methods to the `Human` class now to make it “rangeable.” A successor of a human will have the same first name but with the addition of Jr. at the end:

```
class Human
  def <=>(other)
    self.first_name <=> other.first_name
  end

  def succ
    successor = self.dup # duplicate current object
    successor.first_name = successor.first_name + " Jr."

    successor
  end
end
```

Notice that the `succ` method should return a new object.

After the code above is written, we can go ahead and use `Human` within ranges:

```
start_human = Human.new("John", "Doe")
end_human = Human.new("John Jr. Jr. Jr.", "Doe")

(start_human..end_human).each { |human| human.introduce }
```

The result of this code is as follows

```
Hi, I'm John Doe
Hi, I'm John Jr. Doe
Hi, I'm John Jr. Jr. Doe
Hi, I'm John Jr. Jr. Jr. Doe
```

### Setter Methods

Setter methods are methods with the equal sign (=) at the end of their name. You can define those to give it the feeling of an attribute (or property on languages like C#). This is actually what's constructed when you define a setter attribute.

I will add a setter method to the Human class, `full_name`, so that users can set it directly without setting the first and last names separately:

```
class Human
  def full_name=(value)
    @first_name, @last_name = value.split(" ")
  end
end

human = Human.new("John", "Doe")
human.full_name = "Shay Friedman"
human.introduce # print "Hi, I'm Shay Friedman"
```

### method\_missing

The `method_missing` method is invoked by Ruby when a call to an undefined method occurs. The method receives as parameters the name of the requested method, the arguments passed to it, and the block attached to the call, if it exists.

I'll now add a `method_missing` implementation to the Human class that will write an error to the screen when an undefined method is invoked:

```
class Human
  def method_missing(method_name, *args, &block)
    puts "Humans can't #{method_name}!"
  end
end

john = Human.new("John", "Doe")
john.bark # Prints to the screen "Humans can't bark!"
```

If the `method_missing` implementation doesn't throw an exception, the caller of the method won't know that a different method than requested was invoked (and it returned the expected result of course). This behavior opens a whole new world of opportunities to dynamically execute operations. It is exploited in various Ruby frameworks (such as Ruby



on Rails) to allow a more readable and intuitive syntax. You'll learn how to take advantage of this method in Chapter 8, "Advanced Ruby."

### **const\_missing**

Like the `method_missing` method, the `const_missing` method is invoked when a call to an undefined constant is made. The requested constant name is passed as a symbol to the method. The method can return a value to the caller. Because constants are class related and not instance related, the `const_missing` method should be defined as a class method as well (otherwise it won't be invoked):

```
class Demo
  def Demo.const_missing(const_symbol)
    puts %Q_The constant "#{const_symbol.to_s}" is undefined_
  end
end

Demo::A_CONSTANT # prints "The constant "A_CONSTANT" is undefined"
```

## **The self Keyword**

`self` is used within classes to refer to the class. The important thing to notice about `self` is that the object it refers to is changed according to the context of its invocation. Inside instance methods, `self` will refer to the current instance of the class. Outside instance methods, `self` will refer to the class object itself (inside class methods for example):

```
class Demo
  attr_accessor :instance_attr

  # On method definition, which is outside an instance method definition, self
  # refers to the class object and makes the method a class method
  def self.my_class_method
    puts "I'm a class method!"
  end

  def my_instance_method
    self.my_class_method # Error! self refers to the instance and my_class_method
                        # is a class method.
    self.instance_attr = 5 # OK! instance_attr is an instance attribute
  end
end
```

## **Visibility Control**

Ruby, being an object-oriented language, makes it available to control the visibility of class objects. There are three levels of visibility: public, protected, and private. Ruby is a bit

unusual with the meaning of these visibility levels as it prefers encapsulation capabilities over strict visibility rules:

- ▶ Methods are public unless they are defined differently. Public methods are visible to everyone from anywhere. The only exception is the `initialize` method, which is made private by default.
- ▶ When a method is declared as private, it is accessible only via the instance methods of the class itself and not by the users of the class. Private methods must be called using only their name (even `self.method_name` won't work). Pay attention that private methods are *also accessible* by subclasses of the class.
- ▶ Protected methods are similar to private methods with two differences; they can be called using `self` and they are accessible by *other* instances of the same class.

The way to define the visibility of class objects is done in two ways. The first is to gather methods with the same visibility to the same scope:

```
class Demo
  def initialize
  end

  # Public methods scope

  def m1
  end

  # Protected methods scope
  protected

  def m2
  end

  # Private methods scope
  private

  def m3
  end
end
```

The second way to do that is to define the visibility after the method creation:

```
def m2
end
def m3
end

protected :m2, :m3
```

Changing the visibility of class methods is a bit different and involves using different keywords. `private_class_method` is used as `private`, and `public_class_method` is used as `public` (the default). There is no way to define a protected class method. The technique of using these keywords is the same as above (scoping or explicit setting):

```
class Demo
  def Demo.m4
  end

  private_class_method :m4
end
```

### WHERE VISIBILITY IS NOT APPLIED

For constants, class variables, and instance variables, visibility is not relevant. Constants will always be public and accessible from outside, class variables are always protected, and instance variables are always private.

## Inheritance

Ruby features the object-oriented principle inheritance. Therefore, a class can extend or modify some behaviors of another class. Ruby doesn't allow a multiple inheritance behavior, which means that a class can inherit from, at most, one class. Instead of multiple inheritance, it is possible to create a tree of inheritance so that the descendant classes can access methods and attributes from the superclass of their superclass (or even higher in the tree).

Inheritance is done with the `<` operator:

```
class DescendantClass < SuperClass
  ...code...
end
```

Note that the `initialize` method is inherited just like every other method. This is why, for example, I've mentioned earlier that it is a common practice to create exception classes with nobody that inherits from the `StandardError` class. These classes are copycats of the `StandardError` class with only a different name. The next code sample contains a descendant of the `Human` class: `Doctor`. This class will describe a human whose occupation is to take care of people's health. After defining the class, it is possible to use it just as it would have been done with the `Human` class:

```
class Doctor < Human
end
john = Doctor.new("John", "Doe")
puts john # prints "Human: John Doe"
```

## Overriding Methods

A vital part of the inheritance principle is modifying the behavior of the superclass in a way that matches the current class approach. In Ruby, overriding methods is done implicitly; the method definition is identical to any other method definition. If the method has the same signature as a method on the superclass, this implementation is used, not the one on the superclass. Notice that the result of this behavior is that in Ruby there is no real way to keep a method from being overridden. For example, I'll update the `Human` class and add a getter method to retrieve the full name and then change the `introduce` method to use it:

```
class Human
  def full_name
    "#{self.first_name} #{self.last_name}"
  end

  def introduce
    puts "Hi, I'm #{full_name}"
  end
end
```

Now, in the `Doctor` class I'll override the `full_name` method:

```
class Doctor < Human
  def full_name
    "Dr. #{self.last_name} #{self.first_name}"
  end
end
```

After that, let's see what happens when we define a `Doctor` instance and call the `introduce` method:

```
john = Doctor.new("John", "Doe")
john.introduce # prints "Hi, I'm Dr. Doe John"
```

The following chain of events led to this output:

- ▶ Ruby dynamically looks for the `introduce` method on the `Doctor` class.
- ▶ The method isn't found, so the search continues to the superclass `Human`.
- ▶ The method is found and invoked.
- ▶ Within the method, there is a call to a method named `full_name`.
- ▶ Ruby searches for a `full_name` method on `Doctor` and finds it.
- ▶ Within the `full_name` method, there are calls to `first_name` and `last_name`.

- ▶ Ruby fails to find those on the `Doctor` class, locates them on the `Human` superclass, and invokes them.
- ▶ `first_name` and `last_name` return their values to the `full_name` method on `Doctor`.
- ▶ The `full_name` method constructs the string “Dr. Doe John” and returns it to the `introduce` method on `Human`.
- ▶ The `introduce` method constructs a string with the `full_name` value—“Hi, I’m Dr. Doe John”—and prints it out to the screen.

### Overriding Private Methods

As mentioned previously, Ruby is a very permissive language. This principle finds expression when trying to override private methods: It is possible without any restriction.

Overriding a private method is done just like regular methods.

### Overriding Class Methods

Class methods are inherited from the superclass like other methods and can be overridden. If the inheriting class does not define an implementation to the class method, the superclass method will run.

The exception with class methods is the ability to explicitly run the superclass implementation.

Consider the following A and B classes:

```
class A
  def A.message
    "I am A"
  end
end
```

```
class B < A
  def B.message
    "I am B"
  end
end
```

In the preceding code, B overrides A’s `message` class method. Let’s extend this sample and add the class method `show_message` to class A:

```
def A.show_message
  puts A.message
end
```

Note that I refer to the A’s `message` method explicitly. As a result, the output of the next line will be “I am A”:

```
B.show_message # prints "I am A"
```

To change this behavior so that the inheritance process will run as expected, you should replace the explicit method with the `self` keyword, which will refer to the current class and therefore overridden method implementations (if any):

```
def A.show_message
  puts self.message
end
```

```
# Now if we re-run the previous sample, we will get the expected result:
B.show_message # prints "I am B"
```

### Invoking Superclass Method Implementation

In some cases, we'll want to add code to the superclass method and not replace it entirely. This can be done via the `super` keyword. When this keyword is executed, it invokes a method with the same name on the superclass.

`super` can be called with or without parameters. It has a unique behavior after it's called without arguments and no parentheses: It invokes the superclass method with the same arguments the current method received.

For example, I will inherit from the `Doctor` class and create a new class: `Dentist`. Then I will extend (and not replace) the `introduce` method to add a relevant welcome message:

```
class Dentist < Doctor
  def introduce
    super
    puts "How are your teeth today?"
  end
end

john = Dentist.new("John", "Doe")
john.introduce # prints "Hi, I'm Dr. Doe John
               #         how are your teeth today?"
```

### Abstract Classes

Ruby doesn't have the usual support for abstract classes that you may be familiar with from languages such as C#. Every class in Ruby can be initialized and used, and the same rule applies to methods, too. When it's defined, it can be exploited.

This is the reason why abstract classes are a bit different. The way to approach the abstract classes issue in Ruby is to consider abstract classes as classes that specify a certain flow while this flow is not implemented yet.

For example, I'll extend the `Doctor` class and add to it a method called `work`. In this method, I'll call an abstract method, `write_prescription`. I call it "abstract" because I will

not implement it within the Doctor class but only in its descendants (for example the Dentist class):

```
class Doctor < Human
  def work
    write_prescription
  end
end

class Dentist < Doctor
  def write_prescription
    ...write a prescription...
  end
end

doc = Doctor.new("John", "Doe")
doc.work # Error! write_prescription is not implemented here

dentist = Dentist.new("John", "Doe")
dentist.work # OK
```

Another useful pattern to follow is to have a method in the base class that throws an exception that tells the user to implement it. This is a way of asking the user to override a method. To do that, just add the following method to the Doctor class:

```
def write_prescription
  raise "write_prescription method is not implemented!"
end
```

### Inheritance Collisions

If you're not familiar with the code of the class you're inheriting from, you might mistakenly override one of its methods (even private ones) or variables and dramatically change the way it works. This can lead to a very big mess when the application is executed.

To avoid this kind of chaos, do not inherit from classes if you are unfamiliar with their code. Even though Ruby allows inheriting from every class, even basic ones, it is a good idea to just use these classes and their public API instead of inheriting from them.

## Duck Typing

Now that you know about Ruby's classes, it's time to learn how objects are treated in Ruby.

From the Ruby perspective, every object *might* be what we're looking for. This is what the term *duck typing* means: If it quacks like a duck and swims like a duck, it must be a duck.

Ruby's duck typing philosophy can be demonstrated easily. Look at the next method:

```
def print_all_values(container)
  container.each { |x| print x }
end
```

This is a simple method. It receives an object, iterates over its elements via the `each` method, and prints every item. Our “duck” here is an object that implements the `each` method. It doesn't matter what the object is, who its superclass is, or what it is for. As long as it implements the `each` method, it is a valid input for the `print_all_values` method:

```
# Arrays are valid
a = ["a", "b", "c"]
print_all_values(a) # Prints "abc"

# Strings are valid too
b = "abc"
print_all_values(b) # Prints "abc"

# Custom classes that implement each are valid as well
class CustomClass
  def each
    yield "a"
    yield "b"
    yield "c"
  end
end
c = CustomClass.new
print_all_values(c) # Prints "abc"
```

We can avoid duck typing by enforcing our method to receive parameters of a specific type. This will be done using the `is_a?` method. In the next example, I enforce the parameter to be an array:

```
def print_all_values(container)
  raise TypeError, "container must be an array" unless container.is_a?(Array)
  container.each { |x| print x }
end

a = ["a", "b", "c"]
print_all_values(a) # Prints "abc"

b = "abc"
print_all_values(b) # Error!
```



This is not a recommended practice, though. The more appropriate approach (but still not widely used) is to make sure that the method we need really exists. This is done via the `respond_to?` method:

```
def print_all_values(container)
  unless container.respond_to?(:each)
    raise TypeError, "container must implement the each method"
  end
  container.each { |x| print x }
end
```

## Modules

Modules, like classes, are containers. They contain classes, methods, and constants. Unlike classes, modules cannot have a superclass, cannot be inherited from, and cannot be instantiated. Modules can also be used as mixins, a technique introduced later in this section.

Defining a module is much like defining a class—starting with the `module` keyword, followed by the module name, and ending with the `end` keyword. Module names must start with a capital letter:

```
module DemoModule
  ...module content...
end
```

### CLASSES AND MODULES

Classes in Ruby consist of the `Class` class. Likewise, modules consist of the `Module` class. The `Class` class inherits from the `Module` class. This means that every class is also a module and can access the module private methods (like `include`, as discussed in relation to mixin later in this section).

## Module-Contained Objects

As mentioned previously, modules can contain different objects. The contained objects are accessed in a different way than normal, a way very similar to class object members (not class instance members).

### Classes

As mentioned earlier, modules can contain classes. The way to do that is to just nest the class definition into the module definition. After a class has been defined within a module, the module name followed by two colons should precede its name to call it:

```

module DemoModule
  class Demo
    end
end

d = Demo.new # Error!
d = DemoModule::Demo.new # OK!

```

### Methods

Methods can be defined within modules just like they can within classes. Namespace modules often have only class methods; mixin modules mostly have only instance methods.

The same rules of class methods apply to module methods—their names should be preceded by the module name or by the `self` keyword:

```

module Earth
  def Earth.days_in_year
    Time.local(Time.now.year, 12, 31).yday
  end
end

puts Earth::days_in_year # print 365 (or 366 on leap years)

```

### Constants

A module can also contain constants. Just as with classes, the constants are accessible through the module:

```

module Earth
  HOURS_PER_DAY = 24
end

puts Earth::HOURS_PER_DAY # prints "24"

```

## Namespaces

Namespaces provide a way to group various logically related objects into a single container. You might find this approach useful as your application gets bigger or when writing a stand-alone component. Using modules as namespaces helps prevent name collisions between classes and methods. For example, the `Human` class can be related to `Earth`, but it can also be related to `Mars` in a whole different way. It is possible to create two different classes with different names, `EarthHuman` and `MarsHuman`, but this will become annoying when you want to inherit from those classes. Eventually, you will find yourself with long-named classes that are hard to follow. This is exactly where defining namespaces comes in. We define two modules, one for `Earth` and one for `Mars`, and inside of them we declare any life form we need:

```
module Earth
  class Human
    ...code...
  end
end
```

```
module Mars
  class Human
    ...code...
  end
end
```

Pay attention that now we can no longer access the Human class like we did before. Human exists within modules, so we have to indicate the module we're referring to:

```
# Supposing that both Human classes receive the same constructor parameters
human = Earth::Human.new("John", "Doe")
alien = Mars::Human.new("Al", "ien")
```

## Mixins

The mixin technique in Ruby is powerful. It provides a way to add functionality to classes. A *mixin* is a module that includes instance methods. After you mix the module into a class, the module methods appear as part of the class.

The `include` method is used to mix in a module. `include` can retrieve multiple module names in one call (comma separated). For example, I will add a `Politeable` module that will provide politeness functionality. Then I will include this functionality to the `Doctor` class:

```
module Politeable
  def thank_you
    puts "Thank you"
  end

  def please
    puts "Please"
  end
end

class Doctor < Human
  include Politeable
end

john = Doctor.new("John", "Doe")
john.thank_you # prints "Thank you"
```

To find out whether a class has been mixed in with a specific module, you should use the `is_a?` method:

```
john.is_a? Politeable # = true
john.is_a? Doctor # = true

john.instance_of? Doctor # = true
john.instance_of? Politeable # = false
```

### Available Ruby Modules for Mixin

Ruby already contains several modules ready for mixin that allow developers to add great new functionality to their classes with ease. The common ones are `Comparable` and `Enumerable`. `Comparable`, after you define the `<=>` operator in your class, will add the implementation of `<` `<=` `>` `>=` `==` and `between?` to the class. `Enumerable`, after you define an `each` method in your class, will add various methods that add a lot of functionality (such as `min`, `max`, `sort`, and `member?`).

We've already defined a comparison (`<=>`) method to the `Human` class earlier that compares the human first names. I also add an `each` method that goes through the human full name by letters:

```
class Human
  def <=>(other)
    self.first_name <=> other.first_name
  end

  def each
    full_name.scan(/./m) { |x| yield x }
  end
end
```

Now I can mix in `Comparable` and `Enumerable` to add a lot of functionality to the `Human` class:

```
class Human
  include Comparable, Enumerable
end

john = Human.new("John","Doe")
john.max # the biggest character in the name = "o"
john.sort # sorts the letters of the name = DJehnoo
john.between? Human.new("J", "Doe"), Human.new("Zohn", "Doe") # = true
john.each_with_index do |value, index|
  print "#{index}#{value}"
end # prints "0J1o2h3n4 5D6o7e"
```

## Summary

In this chapter, you learned how to use Ruby to convert arbitrary lines of code into logical, “living” objects. The chapter covered methods, blocks, classes, and modules—the basics that Ruby programmers use every day.

With the knowledge you currently have of the Ruby language, you can develop large modular applications. The next chapters describe more advanced mechanisms that may prove to be lifesavers in various scenarios.

## CHAPTER 7

# The Standard Library

In the process of development, you sometimes hit the point when you feel you're inventing the wheel once again. Your feeling is somewhat justified because most development tasks involve similar utilities (for example, reading from and saving to files, parsing text, sending data, and so forth).

The Ruby language, in addition to the core elements, is distributed with an additional set of libraries that provide tools for executing common tasks easily. Among them are libraries for CSV, SOAP, unit tests, numbers, date and time, HTTP, POP3, RSS, and more. These libraries can save plenty of development hours, and it is a good idea to become familiar with the available features.

For detailed information about all the standard libraries, visit the ruby-doc website: <http://www.ruby-doc.org/stdlib>.

## Using the Libraries

The standard library is not part of the core of Ruby. Therefore, to use one of the libraries, we will have to require it. It will be done using the `require` method mentioned previously in Chapter 5, "Ruby Basics."

For example, if we want to use the `BigDecimal` library, which supports large decimals, we require it first and then use it:

```
require 'bigdecimal'  
12312321434.32432432434*50_000.5 # = 615622227876933.0
```

### IN THIS CHAPTER

- ▶ Using the Libraries
- ▶ Libraries Available in IronRuby
- ▶ Libraries Reference
- ▶ Finding More Libraries

## Libraries Available in IronRuby

On its first version, IronRuby will not contain all of the libraries available for Ruby. Table 7.1 briefly describes the libraries included in IronRuby.

TABLE 7.1 Standard Libraries Available in IronRuby 1.0

| Library    | Description  |
|------------|--|
| Abbrev     | Provides a method that calculates the available abbreviations of a given set of words. Good for calculating suggestions for a given string.                                |
| Base64     | Provides methods for base64 encoding and decoding.   |
| Benchmark  | Provides methods to benchmark Ruby code with detailed reports.   |
| BigDecimal | Provides support for very large or precise decimal numbers.  |
| CGI        | A Common Gateway Interface (CGI) implementation. Provides several classes and methods to read and create HTTP requests and responses.                                      |
| Complex    | Provides support for complex numbers, such as $1+2i$ .   |
| Csv        | Provides comma-separated value (CSV) data creation and reading.  |
| Date       | Provides enhanced date and time operations. For example, converting between the Gregorian and Julian calendars or easy-to-use date enumeration methods.                    |
| Debug      | Provides enhanced debugging capabilities.  |
| Delegate   | Provides a way to inherit from every Ruby class. It acts like a proxy—adding method calls to the delegated class on the delegating class.                                  |
| Digest     | Cryptography support. Provides classes to cipher strings with MD5, SHA256, SHA384, and SHA512 algorithms.  |
| Drb        | Drb stands for dRuby or Distributed Ruby. Like <i>remoting</i> in .NET, Drb allows using objects from other ruby processes located on the same machine or a different one. |
| E2mmap     | E2mmap stands for Exception to Message Mapper. Provides a way to couple exception types with message formats. New exception types can be created with it, too.             |
| English    | Aliases Ruby's strangely named global variables to more readable names.  |
| Erb        | Provides a templating system. Erb makes it very simple to integrate Ruby code in text or HTML files.   |
| Eregexp    | Extends the Regexp class and provides methods to compare a single string against multiple regular expressions at once.   |
| FileUtils  | Provides enhanced file operations (for example, copying and moving files or deleting multiple files simultaneously).   |

TABLE 7.1 Standard Libraries Available in IronRuby 1.0

| Library     | Description  |
|-------------|--|
| Find        | Provides methods to get all files within a directory and all of its subdirectories.  |
| Forwardable | Provides support to the delegate pattern. It allows exposing inner object methods as part of the class itself.   |
| Generator   | Converts an internal iterator to an external iterator. For example, an object that implements each to iterate through a collection can support the end?, next, and current methods with the help of the Generator library. |
| GetOptLong  | Provides an easy-to-use command-line arguments interpreter.  |
| GServer     | Provides a generic server implementation, including thread pool management, simple logging, and multiserver management.  |
| IPAddr      | Provides IP address related methods. Allows you to manipulate an address and to test it against a single IP address or a range of addresses.   |
| JCode       | Provides Japanese strings support.   |
| Kconv       | Provides methods for Kanji string conversion to several encodings.   |
| Logger      | Provides a logging support. Supports different logging levels and autorolling of log files.  |
| MailRead    | An Internet mail message (RFC 822 and RFC 2822) parser.  |
| MathN       | Enhanced mathematical operations (for example, the Prime class for prime numbers support, finding an integer's prime factors, power rational numbers, and finding the square root of a complex number).                    |
| Matrix      | Provides matrix and vector support via the Matrix and Vector classes.  |
| Monitor     | A thread synchronization mechanism.  |
| Mutex_m     | Allows an object or a class to add mutex-like features.  |
| Net/ftp     | Provides support for FTP   |
| Net/ftptls  | Provides support for secured FTP, running with TLS.  |
| Net/http    | Provides support the HTTP  |
| Net/https   | Provides support for secured HTTP, HTTPS.  |
| Net/imap    | Provides support for IMAP  |
| Net/pop     | Provides support for POP3.   |
| Net/smtp    | Provides support for SMTP  |
| Net/telnet  | Provides support for the Telnet protocol.  |
| Net/telnets | Provides support for the secured Telnet protocol.  |
| Observer    | Provides an easy way to implement the observer design pattern.   |



TABLE 7.1 Standard Libraries Available in IronRuby 1.0

| Library     | Description   |
|-------------|---|
| Open-uri    | Provides HTTP, HTTPS, and FTP streams' reading operations.  |
| Open3       | Provides access to the STDIN, STDOUT and STDERR of another application.   |
| OpenSSL     | The SSL protocol implementation. Based on the OpenSSL project.  |
| Optparse    | Provides a command-line arguments interpreter. More enhanced than the GetOptLong library.   |
| OStruct     | Provides a way to define a data object with arbitrary attributes.   |
| Parse_Tree  | Generates an array, which represents the expression tree generated from a given class.  |
| ParseDate   | Provides methods to parse a string and convert it to an array of date parts.  |
| PathName    | Provides support for manipulating file paths (for example, joining paths or using relative paths).  |
| Ping        | Provides a method for testing a connection to a server on a specific port.  |
| PP          | PP stands for PrettyPrinter. It provides a mechanism that allows a more human-readable output when inspecting objects.  |
| PrettyPrint | Provides a way to construct an output and format it according to specified attributes.  |
| PStore      | Provides an easy way to store and read hash data (key-value pairs) in the file system.  |
| Racc/parser | A complicated text parser using a defined grammar for the parsing operation. Can be used to parse very complex content.   |
| Rational    | Provides support for rational numbers (1/2 and not 0.5). After requiring this library, some interactions between numbers will return a Rational object rather than the regular return type. |
| Readbytes   | Extends the IO class and adds a readbytes method to read fixed-length buffers.  |
| Rexml       | A fast and easy-to-use XML processor.   |
| RSS         | An RSS feed reader and writer.  |
| Scanf       | Adds a C-like scanf method to the String and IO classes and to the Kernel module.   |
| Set         | Provides support to set collections (an unordered collection with no duplicates).   |
| Shell       | Enhances the Ruby environment with UNIX commands like pipes.  |

TABLE 7.1 Standard Libraries Available in IronRuby 1.0

| Library | Description  |
|---------|--|
| YAML    | Provides support to serialize and deserialize objects to and from YAML syntax. |
| Zlib    | Provides GZip support.   |

## Libraries Reference

Table 7.1 listed the available libraries. Now this section takes a closer look at the commonly used libraries and some sample uses.

### Abbrev

The abbrev library generates an unambiguous abbreviation list for every given string. It can be used for word-completion processes.

When the abbrev library is included, the `Abbrev::abbrev` method becomes available as well as an `abbrev` method on the `Array` class:

```
require 'abbrev'

abbrevs = Abbrev::abbrev(["IronRuby", "IronPython"])
# abbrevs now contains:
# { "IronP" => "IronPython", "IronPy" => "IronPython",
#   "IronPyt" => "IronPython", "IronPyth" => "IronPython",
#   "IronPytho" => "IronPython", "IronPython" => "IronPython",
#   "IronR" => "IronRuby", "IronRu" => "IronRuby",
#   "IronRub" => "IronRuby", "IronRuby" => "IronRuby" }

# Identical call
abbrevs = ["IronRuby", "IronPython"].abbrev
```

### Base64

The base64 library is used to encode or decode strings in the base64 textual encoding:

```
require 'base64'
# Encode:
str = Base64.encode64("Hello World")
puts str # prints "SGVsbG8gV29ybGQ="
# Decode:
decoded_str = Base64.decode64(str)
puts decoded_str # prints "Hello World"
```

## Benchmark

The benchmark library proves very handy when you need to measure how long it takes for parts of the code to execute.

There are a few ways to benchmark code. The first is to measure a single block:

```
require 'benchmark'

result = Benchmark.measure { 10000.times { x = "IronRuby" } }
puts result # Prints " 0.202801  0.000000  0.202801 ( 0.114000)"
```

The result format is as follows:

```
<user CPU time> <system CPU time> <total time> ( <elapsed real time>)
```

Another way is to measure multiple code blocks sequentially (with an optional parameter of a block caption):

```
require 'benchmark'

# The 7 parameter is the length of the caption
Benchmark.bm(7) do |b|
  b.report("times") { 10000.times { x = "IronRuby" } }
  b.report("range") { (1..10000).each { x = "IronRuby" } }
  b.report("upto") { 1.upto(10000) { x = "IronRuby" } }
end
```

The preceding code outputs to the screen the next summary:

|       | user     | system   | total    | real        |
|-------|----------|----------|----------|-------------|
| times | 0.109201 | 0.000000 | 0.109201 | ( 0.087000) |
| range | 0.062400 | 0.000000 | 0.062400 | ( 0.047000) |
| upto  | 0.015600 | 0.000000 | 0.015600 | ( 0.020000) |

If more accurate results are needed, consider using the `bmbm` method rather than `bm`. `bmbm` stands for “benchmark benchmark,” and it does exactly that—it runs the code blocks twice. The first run is a “rehearsal” mode, and the second run is the “real” mode. This `bmbm` method removes the possibility that the loading time of the interpreter will be included in the results.

## BigDecimal

Ruby has support for floating-point numbers via the `Float` class. The problem with its built-in support is the use of binary floating-point arithmetic. A binary floating-point representation is good for 0.5 and 0.25 but isn’t that good for 0.1, 0.01, and so on. This

leads to problems with the precision of Float variables. For example, the next code line returns false:

```
(0.7-0.5) == 0.2
```

The BigDecimal library uses decimal arithmetic, which solves the precision issues; BigDecimal numbers support more than a billion digits after the floating point.

Using BigDecimal numbers resolves the previous issue:

```
require "bigdecimal"  
(BigDecimal.new("0.7") - BigDecimal.new("0.5")) == BigDecimal.new("0.2") #=> true
```

Creating a BigDecimal number is done via its new method. The decimal value is passed to the constructor as a string:

```
dec = BigDecimal.new("0.34234324442565634632")  
dec = BigDecimal("0.34234324442565634632") # Same as above - new is not required
```

BigDecimal also adds support for infinity. A BigDecimal can be initialized using "Infinity" or "-Infinity" as a value:

```
pos_inf = BigDecimal.new("Infinity")  
neg_inf = BigDecimal.new("-Infinity")
```

As a result of its infinity support, you should notice that dividing by zero is acceptable with BigDecimals, and the result will be infinity.

## Complex

The complex standard library adds complex numbers support to the Ruby language.

Defining a complex number is done with or without the new method:

```
require "complex"  
c = Complex.new(1,-2) # = 1 - 2i  
c = Complex(1,-2) # = 1 - 2i
```

After a complex number exists, all arithmetic operators are available for use. The Numeric and Math classes are also updated to support complex numbers.

## CSV

CSV (Comma-Separated-Values) is a common format used to present column-based data. For example, data that consist of first name and last name fields will be presented in CSV format as "Shay, Friedman" or "John, Doe".

The CSV library adds support for creating and parsing comma-separated values (CSV) data.

Creating and reading CSV files is incorporated into the library:

```
require "csv"
# Create a CSV file
CSV.open("data.csv", "w") do |w|
  w << ["New York City", "New York", "USA"]
  w << ["Toronto", "Ontario", "Canada"]
end
# Generates a file with two lines:
# New York City,New York,USA
# Toronto,Ontario,Canada

# Read a CSV file
CSV.open("data.csv", "r") do |row|
  row # 1st iteration = ["New York City", "New York", "USA"]
  # 2nd iteration = ["Toronto", "Ontario", "Canada"]
end
```

However, files are not a necessity. A CSV string can be parsed, too:

```
require "csv"
str = "IronRuby Unleashed,Shay Friedman"
CSV.parse_line(str) # = ["IronRuby Unleashed", "Shay Friedman"]
```

## Digest

The digest library is a cryptography library supporting MD5 and SHA encryption algorithms.

The method `hexdigest` exists in all the different encryption classes and is used to encrypt a string using the classes' encryption algorithm.

To use MD5 encryption, the digest/MD5 library should be required. For SHA encryption, the digest/SHA2 library should be used:

```
require "digest/MD5"
Digest::MD5.hexdigest("Hi") # = "c1a5298f939e87e8f962a5edfc206918" (32 chars)

require "digest/SHA2"
Digest::SHA256.hexdigest("Hi") # = "3639efcd08abb273b1619e82e78c29a7df02c1..."
➡(64 chars)
Digest::SHA384.hexdigest("Hi") # = "efcbca4c3e81ba9f55cfc49bc8bf20d4b0e254..."
➡(96 chars)
Digest::SHA512.hexdigest("Hi") # = "45ca55ccaa72b98b86c697fdf73fd364d4815..."
➡(128 chars)
```

## E2MMAP

The `e2mmap` library is an exception-mapping library. It allows coupling of messages with exception types. For example, the `TypeError` can be coupled with the message “Wrong type!” After they are attached, raised `TypeError`s always results in a “Wrong type!” message:

```
require "e2mmap"
class MyMath
  extend Exception2MessageMapper
  def_e2message TypeError, "Wrong type!"

  def double(a)
    Raise TypeError unless a.is_a?(Numeric)
    a*2
  end
end
```

```
MyMath.new.double("hi") # TypeError occurs with message "Wrong type!"
```

Note that to raise `e2mmap` exceptions, the `Raise` method is used rather than the regular `raise` method.

### Dynamic Messages

The library also allows adding dynamic content to the messages, which will be provided after the exception has been raised. For example, I'll change the preceding sample to support a dynamic error message:

```
require "e2mmap"
class MyMath
  extend Exception2MessageMapper
  def_e2message TypeError, "Type %s isn't allowed here!"

  def double(a)
    Raise TypeError, a.class unless a.is_a?(Numeric)
    a*2
  end
end

MyMath.new.double("hi")
# TypeError occurs with message "Type String isn't allowed here!"
```

### Defining New Exception Types

`E2mmap` also provides a way to easily define new exception types:

```

require "e2mmap"
class MyMath
  extend Exception2MessageMapper

  # def_exception exception_name, message, superclass (optional)
  def_exception :MyMathError, "%s isn't allowed", StandardError

  def double(a)
    Raise MyMathError, a.class unless a.is_a?(Numeric)
    a*2
  end
end

MyMath.new.double("hi") # MyMathError occurs with message "String isn't allowed"

```

## English

The English standard library aims to make Ruby code more human readable. We've seen in several cases that Ruby has global variables that define behaviors across the language. For example, `$!` is the last error object. After you require the English library, this variable can be accessed via `$ERROR_INFO`:

```

require 'english'
begin
  5/0
rescue
  $!.message # OK
  $ERROR_INFO.message # Also OK!
end

```

Table 7.2 lists all the global variable aliases provided by the English standard library.

TABLE 7.2 English Library Global Variable Aliases

| Global Variable  | Alias                                 |
|------------------|---------------------------------------|
| <code>\$!</code> | <code>\$ERROR_INFO</code>             |
| <code>\$@</code> | <code>\$ERROR_POSITION</code>         |
| <code>\$;</code> | <code>\$FS</code>                     |
| <code>\$;</code> | <code>\$FIELD_SEPARATOR</code>        |
| <code>\$,</code> | <code>\$OFS</code>                    |
| <code>\$,</code> | <code>\$OUTPUT_FIELD_SEPARATOR</code> |

TABLE 7.2 English Library Global Variable Aliases

| Global Variable      | Alias                                  |
|----------------------|--|
| <code>\$/</code>     | <code>\$RS</code>                      |
| <code>\$/</code>     | <code>\$INPUT_RECORD_SEPARATOR</code>  |
| <code>\$\</code>     | <code>\$ORS</code>                     |
| <code>\$\</code>     | <code>\$OUTPUT_RECORD_SEPARATOR</code> |
| <code>\$.</code>     | <code>\$INPUT_LINE_NUMBER</code>       |
| <code>\$.</code>     | <code>\$NR</code>                      |
| <code>\$_</code>     | <code>\$LAST_READ_LINE</code>          |
| <code>\$&gt;</code>  | <code>\$DEFAULT_OUTPUT</code>          |
| <code>\$&lt;</code>  | <code>\$DEFAULT_INPUT</code>           |
| <code>\$\$</code>    | <code>\$PID</code>                     |
| <code>\$\$</code>    | <code>\$PROCESS_ID</code>              |
| <code>\$?</code>     | <code>\$CHILD_STATUS</code>            |
| <code>\$~</code>     | <code>\$LAST_MATCH_INFO</code>         |
| <code>\$=</code>     | <code>\$IGNORECASE</code>              |
| <code>\$*</code>     | <code>\$ARGV</code>                    |
| <code>\$&amp;</code> | <code>\$MATCH</code>                   |
| <code>\$`</code>     | <code>\$PREMATCH</code>                |
| <code>\$'</code>     | <code>\$POSTMATCH</code>               |
| <code>\$+</code>     | <code>\$LAST_PAREN_MATCH</code>        |

## Erb

Erb stands for eRuby and is a template engine. It allows inserting dynamic code into text strings.

The syntax of an Erb template is similar to classic ASP syntax. To execute code, the code should be written inside `<% %>`. To add an expression result to the page, the `<%= %>` should be used.

For example, Erb can be used to create an HTML page (also known as RHTML):

```
require "erb"
template = <<EOF
<HTML>
```



```

<BODY>
  <% if welcome %>
    Welcome <%=name%>!
  <% else %>
    Goodbye <%=name%>!
  <% end %>
</BODY>
</HTML>
EOF

```

```

html = ERB.new(template)

# Set the needed values for the template
welcome = true
name = "John Doe"
# Print the result
puts html.result

```

The result of this code is the following HTML syntax printed to the screen:

```

<HTML>
  <BODY>

    Welcome John Doe!

  </BODY>
</HTML>

```

The `result` method can also receive a binding object as a parameter to use variables and methods of another object:

```

require "erb"

class Human
  attr_accessor :first_name, :last_name

  def get_binding
    binding
  end
end

h = Human.new
h.first_name = "John"
h.last_name = "Doe"

```

```
text = ERB.new("<% 3.times do %><%=first_name%> <%=last_name%> <% end %>")
text.run(h.get_binding) # run is like result and just prints the result, too.
```

The result is “John Doe John Doe John Doe.”

## FileUtils

The FileUtils library enhances file operations in Ruby. For example, it allows copying and moving files, deleting a directory with its content, deleting multiple files and directories, and more.

For more information, see the “Handling Files” section in Chapter 8, “Advanced Ruby.”

## Logger

The logger standard library provides a logging system for Ruby applications. A log can be written to any stream or directly to a filename:

```
require "logger"
Logger.new(STDOUT) # Write to the default stdout stream
Logger.new("c:/logfile.log") # Write to C:\logfile.log file
```

### Adding Log Entries

Adding a log entry is done with the log entry level (FATAL, ERROR, WARN, INFO, DEBUG, and UNKNOWN):

```
require "logger"
log = Logger.new("c:/logfile.log") # Write to C:\logfile.log file

log.info { "Started" }
log.fatal { "Oh no!" }
# Every log can also be sent its associated progname:
log.debug("My class") { "this is debug data!" }
log.info("Global") { "info!" }
log.unknown { "Ended" }

# Closing the log
log.close
```

### Logger Level

The entire logger also has a level, and if the logger level is higher than the entry level, the entry is ignored.

The levels priority, from bottom to top, are DEBUG > INFO > WARN > ERROR > FATAL > UNKNOWN:

```
require "logger"
log = Logger.new("c:/logfile.log")
log.level = Logger::INFO # Set the log level
```

```
log.info { "Started on #{Time.now}" }
log.debug { "this is debug data!" } # Ignored
log.fatal { "oh no!" }
log.error { "error!" }
log.unknown { "Ended" }
```

### Automatic Log Rotation

The logger library also allows automatic rotation of log files. When creating a log, you can define whether it will be rotated daily, weekly, or monthly:

```
log = Logger.new("c:/logfile.log", "daily")
log = Logger.new("c:/logfile.log", "weekly")
log = Logger.new("c:/logfile.log", "monthly")
```

If day rotation is not suitable, it can be done also when a file hits a specified size. When the size is specified, it is also possible to define how many log files should be stored:

```
# Rotate log file when it is 1Mb in size and delete log files older than 14 days:
log = Logger.new("c:/logfile.log", 14, 1024)
```

## Monitor

The monitor standard library adds another thread synchronization mechanism. Monitors are similar to mutexes with a few differences, such as the capability to incorporate monitors into objects.

For more information about the monitor standard library, see the “Threads” section in Chapter 8.

## Net/http

The net/http standard library contains classes and methods to use HTTP. It allows retrieving pages from web servers, post form data, send requests, and get responses.

### Retrieving Pages

The library enables retrieving pages in a very simple and direct way:

```
require "net/http"
str = Net::HTTP.get "www.ironruby.net", "/About"
File.open("ruby.net.txt", "w") { |file| file.print str }
```

The preceding code retrieves the HTML content of the given page and writes it into a file. This operation can be done in a more customized way, specifying the port and the request, for instance:

```
require "net/http"
require "uri" # URI is another standard library explained later in this chapter.

url = URI.parse('http://www.ironruby.net/About')

# Define a GET request
req = Net::HTTP::Get.new(url.path)
# Start an HTTP communication
res = Net::HTTP.start(url.host, url.port) {'http'
  # Send the request
  http.request(req)
}
#Print the response headers
res.each_header { |key, value| puts "#{key} = #{value}" }
# Print the response content
puts res.body
```

### Posting Form Data

Instead of reading only data, the library features a more active approach. It features methods for posting form data to web pages. For example, in the following sample I'll post a search query to Google Blog Search:

```
require 'net/http'
require 'uri'

res = Net::HTTP.post_form(URI.parse('http://blogsearch.google.com/blogsearch'),
  {"q" => "IronRuby"})
puts res.body
```

### Observer

Observer is a well-known and commonly used design pattern. The observer standard library makes it easy to incorporate this pattern into classes.

See the “Design Patterns” section in the next chapter for more information about the observer design pattern and the observer standard library.

### Open-uri

The open-uri standard library provides an easy-to-use support for handling HTTP, HTTPS, and FTP streams.

After you open such a stream, it is possible to read it or retrieve its metadata like the content type or last modification date:

```
require "open-uri"
open("http://www.ironruby.net") do |stream|
  stream.read          # page content
  stream.base_uri      # = URI object of "http://www.ironruby.net"
  stream.content_type  # = "text/html"
  stream.charset       # = "utf-8"
  stream.content_encoding # = []
  stream.last_modified # = nil
end
```

The `open` method can also receive header information for its request:

```
require "open-uri"
open("http://www.ironruby.net",
     "UserAgent" => "IronRuby",
     "Referer" => "http://www.ironshay.com"
     # ...any other header information...
     ) do |stream|
  stream.read
end
```

### Proxies

`Open-uri` uses the default proxies, which are defined on the environment variables `http_proxy`, `https_proxy`, and `ftp_proxy`. If you need to use a different proxy or not use a proxy at all, the `proxy` option should be passed to the `open` method:

```
require "open-uri"
open("http://www.ironruby.net",
     :proxy => "http://proxy.mynetwork.net" # Set the proxy manually
     ) { |stream| stream.read }

open("http://www.ironruby.net",
     :proxy => nil # Do not use any proxy server
     ) { |stream| stream.read }
```

### Progress

The `open` method also allows passing it a `proc` or a `lambda` that will receive updates about the transferred size. Two arguments can be helpful for that: `content_length_proc` and `progress_proc`.

The `content_length_proc` `proc` is invoked before the transfer begins, passing the expected content length to the `proc`. It is possible that the length would be `nil` (when the requested URI doesn't support it).

The `progress_proc` is invoked after each fragment is returned from the server with the transferred size. For example, the following code outputs on the screen the download progress in percentage:

```
require "open-uri"
content_length = 0
stream = open("http://www.ironshay.com",
  :content_length_proc => lambda { |length|
    content_length = length unless length == nil; puts "Starting..." },
  :progress_proc => lambda { |size|
    puts "#{((size.to_f/content_length.to_f)*100).to_i}% done" }
)
stream.close
```

## Ping

The ping standard library enables pinging a given server on a specific port. This standard library adds a single method, `Ping.pingecho`, which does the actual ping request.

For example, the next statement tries to open a connection to the Echo service port:

```
require "ping"
Ping.pingecho("127.0.0.1")
```

The `pingecho` method also supports two more arguments, `timeout` (by seconds) and `service name/port number`:

```
require "ping"
# Tries to ping ironruby.net on port 80 with timeout of 10 seconds
Ping.pingecho("ironruby.net", 10, 80)
```

### PING LIBRARY OPERATION

Note that the ping operation is not the regular ICMP request. The library's ping operation is done by trying to open a connection to a port.

Instead of the port number, a service name can be passed. Table 7.3 lists the available socket services.

TABLE 7.3 Available Socket Services

| Service Name | Port Number |
|--------------|-------------|
| "echo"       | 7           |
| "discard"    | 9           |
| "systat"     | 11          |
| "daytime"    | 13          |
| "netstat"    | 15          |
| "qotd"       | 17          |
| "chargen"    | 19          |
| "ftp-data"   | 20          |
| "ftp"        | 21          |
| "telnet"     | 23          |
| "smtp"       | 25          |
| "time"       | 37          |
| "rlp"        | 39          |
| "name"       | 42          |
| "whois"      | 43          |
| "domain"     | 53          |
| "nameserver" | 53          |
| "mtp"        | 57          |
| "bootp"      | 67          |
| "tftp"       | 69          |
| "rje"        | 77          |
| "finger"     | 79          |
| "http"       | 80          |
| "link"       | 87          |
| "supdup"     | 95          |
| "hostnames"  | 101         |
| "iso-tsap"   | 102         |
| "dictionary" | 103         |

TABLE 7.3 Available Socket Services

| <b>Service Name</b> | <b>Port Number</b> |
|---------------------|--------------------|
| "x400"              | 103                |
| "x400-snd"          | 104                |
| "csnet-ns"          | 105                |
| "pop"               | 109                |
| "pop2"              | 109                |
| "pop3"              | 110                |
| "portmap"           | 111                |
| "sunrpc"            | 111                |
| "auth"              | 113                |
| "sftp"              | 115                |
| "path"              | 117                |
| "uucp-path"         | 117                |
| "nntp"              | 119                |
| "ntp"               | 123                |
| "nbname"            | 137                |
| "nbdatagram"        | 138                |
| "nbsession"         | 139                |
| "NeWS"              | 144                |
| "sgmp"              | 153                |
| "tcprepo"           | 158                |
| "snmp"              | 161                |
| "snmp-trap"         | 162                |
| "print-srv"         | 170                |
| "vmnet"             | 175                |
| "load"              | 315                |
| "vmnet0"            | 400                |
| "sytek"             | 500                |
| "biff"              | 512                |



TABLE 7.3 Available Socket Services

| <b>Service Name</b> | <b>Port Number</b> |
|---------------------|--------------------|
| "exec"              | 512                |
| "login"             | 513                |
| "who"               | 513                |
| "shell"             | 514                |
| "syslog"            | 514                |
| "printer"           | 515                |
| "talk"              | 517                |
| "ntalk"             | 518                |
| "efs"               | 520                |
| "route"             | 520                |
| "timed"             | 525                |
| "tempo"             | 526                |
| "courier"           | 530                |
| "conference"        | 531                |
| "rxd-control"       | 531                |
| "netnews"           | 532                |
| "netwall"           | 533                |
| "uucp"              | 540                |
| "klogin"            | 543                |
| "kshell"            | 544                |
| "new-rwho"          | 550                |
| "remotefs"          | 556                |
| "rmonitor"          | 560                |
| "monitor"           | 561                |
| "garcon"            | 600                |
| "maitrd"            | 601                |
| "busboy"            | 602                |
| "acctmaster"        | 700                |

TABLE 7.3 Available Socket Services

| <b>Service Name</b> | <b>Port Number</b> |
|---------------------|--------------------|
| "acctslave"         | 701                |
| "acct"              | 702                |
| "acctlogin"         | 703                |
| "acctprinter"       | 704                |
| "elcsd"             | 704                |
| "acctinfo"          | 705                |
| "acctslave2"        | 706                |
| "acctdisk"          | 707                |
| "kerberos"          | 750                |
| "kerberos_master"   | 751                |
| "passwd_server"     | 752                |
| "userreg_server"    | 753                |
| "krb_prop"          | 754                |
| "erlogin"           | 888                |
| "kpop"              | 1109               |
| "phone"             | 1167               |
| "ingreslock"        | 1524               |
| "maze"              | 1666               |
| "nfs"               | 2049               |
| "eklogin"           | 2105               |
| "rmt"               | 5555               |
| "mtb"               | 5556               |
| "man"               | 9535               |
| "w"                 | 9536               |
| "mantst"            | 9537               |
| "bnews"             | 10000              |
| "rscs0"             | 10000              |
| "queue"             | 10001              |

TABLE 7.3 Available Socket Services

| Service Name | Port Number |
|--------------|-------------|
| "rscs1"      | 10001       |
| "poker"      | 10002       |
| "rscs2"      | 10002       |
| "gateway"    | 10003       |
| "rscs3"      | 10003       |
| "remp"       | 10004       |
| "rscs4"      | 10004       |
| "rscs5"      | 10005       |
| "rscs6"      | 10006       |
| "rscs7"      | 10007       |
| "rscs8"      | 10008       |
| "rscs9"      | 10009       |
| "rscsa"      | 10010       |
| "rscsb"      | 10011       |
| "qmaster"    | 10012       |

## Rational

The rational standard library adds support for rational numbers. A rational number is a number that is expressed as a fraction  $p/q$ , where  $p$  and  $q$  are integers and  $q$  is not zero. This means that 0.5 is expressed as  $1/2$  and not as 0.5.

Creating a rational number is done with the class name:

```
require "rational"
Rational(7, 2) # = 7/2
Rational(5) # = 5/1
```

When you create rational numbers this way, you also reduce the number to its lowest term:

```
Rational(11, 22) # = 1/2 (not 11/22)
```

To prevent this reduction, use the `new!` method to create the number:

```
Rational.new!(11, 22) # = 11/22
```

Rational numbers support all the regular arithmetic operators. In addition, after the rational library is required, every integer number can be converted to a rational number via the `to_r` method.

## Rexml

Rexml is an XML processor supporting files, strings, and streams. It features methods to iterate through the XML or to search it via XPath. In addition to parsing existing XML documents, the rexml library supports generating XML documents, too.

### Generating XML Documents

There are two ways of building an XML document with rexml. The first is by passing it the whole XML as text:

```
require "rexml/document"

xml = <<EOF
<Countries>
  <Country Name="USA">
    <Population>306566000</Population>
    <Continent>America</Continent>
  </Country>
  <Country Name="Ireland">
    <Population>5981448</Population>
    <Continent>Europe</Continent>
  </Country>
</Countries>
EOF

doc = REXML::Document.new xml
```

The second way is to actually generate the XML elements by code. The next code sample generates the same XML document as the preceding one:

```
require "rexml/document"
doc = REXML::Document.new
root = doc.add_element "Countries"
usa = root.add_element "Country", {"Name"=>"USA"}
usa.add_element("Population").text = "306566000"
# add_text sets the text or appends if text already exists
usa.add_element("Continent").add_text("America")
ireland = root.add_element "Country", {"Name"=>"Ireland"}
# adds a text node explicitly
ireland.add_element("Population").add(REXML::Text.new("5981448"))
# Same as the add method
ireland.add_element("Continent") << REXML::Text.new("Europe")
```

**Reading XML Documents**

Reading an XML document can also be done in two different ways. The first is by iterating through the elements. The examples are based on the XML document defined in the previous samples:

```
def print_element(elem)
  # Print the beginning of the node
  print "<" + elem.name
  # Print the attributes
  elem.attributes.each { |name, value| print " #{name} = \"#{value}\" " }
  # Close the node beginning
  puts ">"
  # Print text is exists
  puts elem.text if elem.has_text?
  # Print all child elements
  elem.elements.each { |child| print_element(child) }
  # Close the node
  puts "</#{elem.name}>"
end
# Start printing from the root
print_element(doc.root)
```

Another way to read the XML document is via XPath. XPath can be used from the XML document object or by itself:

```
# Print all Country elements where their "Name" attribute equals "USA"
doc.elements.each("//Country[@Name = 'USA']") { |elem| puts elem }
# With the XPath class, same as above
REXML::XPath.each(doc, "//Country[@Name = 'USA']") { |elem| puts elem }
```

**Singleton**

The singleton standard library adds a singleton module, which after it is included makes it a singleton class, which follows the singleton design pattern.

See the “Design Patterns” section in Chapter 8 for more information about the singleton design pattern and the singleton standard library.

**Socket**

The various socket libraries feature different ways to communicate with other hosts via the network.

To use one of the socket libraries, "socket" should be required:

```
require "socket"
```

The socket library has a hierarchy of inheritance, as shown by the inheritance tree in Figure 7.1.

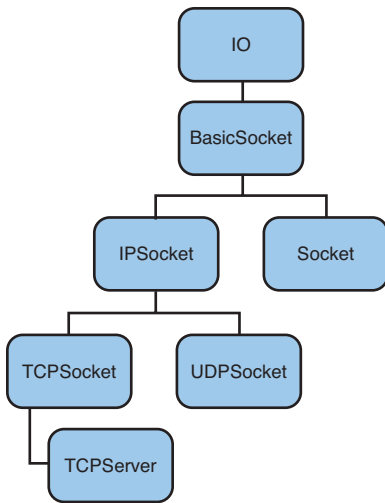


FIGURE 7.1 The inheritance tree of the socket standard libraries.

### BasicSocket

BasicSocket is an abstract class that all the other socket libraries inherit from.

### Socket

The Socket class provides access to the operating system implementation of socket. It features methods for obtaining address information.

For example, fetching the local host name is easy:

```
require "socket"
Socket.gethostname # = the host name
```

Another handy method is `gethostbyname`. When provided with a hostname or an IP address, it returns an array with information about the host:

```
require "socket"
arr = Socket.gethostbyname("google.com")
arr[0] # = hostname
arr[1] # = An array of aliases to the hostname (if any)
arr[2] # = The address family. Usually one of the
      # constants Socket::AF_INET (network) or Socket::AF_INET6 (V6 network)
arr[3] # = The binary value of the address
```

**IPSocket**

IPSocket is the base class for socket classes that base their transportation on IP.

**UDPSocket**

UDP is a stateless protocol with a “send and forget” philosophy. The UDPSocket class can be used to send and receive UDP packets.

The simplest way to send or receive UDP messages is to bind the socket to a specific port and then use `send` to send the message or `recvfrom` and `recvfrom_nonblock` to receive the message:

```
require "socket"
# Create the listening socket
listening = UDPSocket.new
listening.bind("127.0.0.1", 7891) # bind to port 7891
# Send a UDP message
UDPSocket.new.send "IronRuby", 0, "127.0.0.1", 7891
# waits to receive a max-64-characters message and print it
puts listening.recvfrom(64)[0] # Prints IronRuby
```

**TCPSocket**

TCP provides a reliable and ordered delivery of a data stream. The TCPSocket class is used to send messages and receive replies. It cannot be used to listen for incoming messages—this is the target of the TCPServer class.

For example, the following code connects to a server on port 9988, sends it a message, and receives one:

```
require "socket"
s = TCPSocket.new("localhost", 9988)
# Write "Hi" to the socket stream
s.puts "Hi"
# Read from the socket
s.gets
s.close
```

**TCPServer**

TCPServer is aimed to accept incoming requests and send responses if needed.

The following code is the server that the TCP socket from the preceding example has connected to. It listens to incoming connections on port 9988, prints the request message onscreen, and returns a message of its own:

```
require "socket"

server = TCPServer.new("localhost", 9988) # Listen on port 9988
while session = server.accept
```

```
# Print the request
puts session.gets
# Send response
session.puts "Hello from server"
# Close the session
session.close
end
```

## Thread

The thread library adds enhanced thread synchronization support. Requiring the thread library adds the `ConditionVariable`, `Mutex`, `Queue`, and `SizedQueue` classes that provide different thread synchronization mechanisms.

See the Threads section in the next chapter for more information about the thread standard library.

## YAML

YAML is a markup language similar in concept to XML. It is designed for serializing objects to a textual representation. The YAML standard library helps in serializing and deserializing objects using YAML.

Read more about the YAML standard library on the “Marshalling” section in the next chapter.

## WEBrick

WEBrick is a HTTP server library. It provides everything you need to create a HTTP-based server or application.

WEBrick is “low level.” You will find creating a normal HTTP server quite difficult. However, numerous libraries built on top of the WEBrick framework make it much easier to use (like Tofu).

WEBrick is all about servlets. Servlets handle requests and responses in WEBrick and can be defined in several ways.

On the next code sample, I create an HTTP server on the default port (80). I also define a servlet to respond to requests to the page `/IronRuby`. When users request this page, they will be welcomed with a message:

```
require "webrick"
# Create the server object
server = WEBrick::HTTPServer.new
# Set behavior when http://localhost/IronRuby is called
server.mount_proc('/IronRuby') do |req, resp|
  resp.body = "<h1>IronRuby Rules!!!</h1>"
end
```



```
end
# start the server
server.start
```

An in-depth coverage of WEBrick is beyond the scope of this book; for more information visit <http://www.webrick.org>.

## WEBRICK USES

WEBrick isn't used for production purposes. It is used mainly for small proof-of-concept projects and testing purposes. Several other similar frameworks are considered more efficient and fast (for example, Mongrel and Thin, which can be used via RubyGems).

## Zlib

Zlib provides support for the file-compression algorithm Gzip. The Zlib standard library has two main classes: `GzipWriter` for writing Gzip files and `GzipReader` for reading files.

### Writing Files

The `GzipWriter` class is responsible for writing files. The writer class compresses the given data and writes the compressed result to a specified file. To do that, a filename should be passed to the `GzipWriter` constructor and an associated code block, which writes the data to the Gzip stream:

```
require "zlib"
# Generate a Gzip file "compressed.gz" with the defined text
Zlib::GzipWriter.open("compressed.gz") do |gz|
  gz.write "This data will be compressed, "
  gz.write "another line of data, "
  gz.write "more data..."
end
```

### Reading Files

To achieve the opposite of `GzipWriter`, you can use the `GzipReader` class. This class decompresses the data and returns the original content:

```
require "zlib"
Zlib::GzipReader.open("compressed.gz") { |gz|
  print gz.read
}
```

This code returns the data we wrote in the preceding example:

```
"This data will be compressed, another line of data, more data..."
```

## Finding More Libraries

The libraries introduced here are just those that ship with IronRuby. Numerous other libraries are available, and most are free and ready to use.

Several websites contain libraries and allow users to search and download them:

- ▶ RubyForge is the biggest libraries resource and is being updated all the time: <http://rubyforge.org>.
- ▶ The Ruby Application Archive is also a good resource for libraries. It is a part of the official Ruby language website: <http://raa.ruby-lang.org>.

## Summary

This chapter introduced you to the available libraries that enhance Ruby's built-in features. As you can see, the standard library contains powerful tools for various different programming tasks, including logging, networking, design patterns, and threading. These tools have been tested and implemented in a generic way to fit different scenarios.

Now that you are aware of the possibilities, you can use these libraries and save yourself the time of implementing them and even make your code more robust and readable.

*This page intentionally left blank*

# CHAPTER 8

## Advanced Ruby

This chapter covers advanced techniques and operations that are available in Ruby. You learn about threads and file operations in Ruby. You also learn about reflection, which is outstanding compared to what you might be familiar with from static languages such as C# and RubyGems (a library container). The chapter finishes by showcasing several design patterns and their implementation in Ruby.

### Threads

Ruby supports multithreaded applications. Therefore, code blocks can be defined as threads and be run in parallel. For example, remember the Human class. Humans, as we know, usually do several tasks at the same time (for example, seeing and hearing). With the help of multithreading, we can implement these methods to run simultaneously.

Just like other programming languages, multithreading is achieved with the Thread class.

Starting a new thread is done with the new or start methods, followed by a code block that will be run in parallel:

```
Thread.new do
  puts "I'm here"
end
```

```
Thread.start do
  puts "I'm also here!"
end
```

### IN THIS CHAPTER

- ▶ Threads
- ▶ Handling Files
- ▶ Reflection
- ▶ Marshaling
- ▶ RubyGems
- ▶ Design Patterns

A thread can return a value. Just like methods, the return value is the last invoked expression on the thread code block. To obtain the thread return value, the `value` method should be called. If the thread hasn't ended yet, the call to `value` is blocking, and the application will hang until the thread is done:

```
t = Thread.new do
  sleep 3 # Sleeps for 3 seconds
  "I'm a thread"
end

puts t.value # Prints "I'm a thread" after 3 seconds
```

Threads in Ruby are not blocking threads. When the main application thread flow is done, the application terminates along with its threads, running or not. To prevent the application from terminating while the threads are still working, a waiting mechanism should be written. This can be done with the `value` method call as we've just seen, by creating an infinite loop or by calling the `join` method, which blocks the current thread until the associated thread terminates:

```
t = Thread.new do
  sleep 3 # Sleeps for 3 seconds
  puts "I'm a thread"
end

t.join # waits until the thread ends
```

The preceding code prints “I’m a thread” to the screen. If I hadn’t called the `join` method, the application would have terminated before the thread could manage to print the message to the screen.

## IRONRUBY THREADS IMPLEMENTATION

IronRuby uses .NET background threads. Consider the next line of code:

```
Thread.new {}
```

This line of IronRuby code is translated to the next C# code:

```
Thread result = new Thread(new ThreadStart(delegate() { }));
result.IsBackground = true;
result.Start();
```

Make sure the thread is generated as a background thread. This is done to conform to Ruby's thread behavior, which doesn't stop the application from terminating unless a waiting mechanism is written.

---

## Exceptions Within Threads

By default, when an exception occurs within a thread's code block, the application doesn't stop. Actually, in such case, the error will be discovered only when `value` or `join` are called. If they aren't called, the error will be "swallowed" without anyone noticing it ever happened:

```
t = Thread.new do
  sleep 3 # Sleeps for 3 seconds
  raise "oh no"
end

t.join # Only now the "oh no" error terminates the application
```

There is a way, however, to change this behavior. The `Thread` class has an attribute `abort_on_exception`. When this attribute is set to `true`, the application terminates if an error occurs within a thread, just like code outside threads.

This attribute can be set for all threads at once using the class attribute `Thread.abort_on_exception`:

```
Thread.abort_on_exception = true

t = Thread.new do
  # The exception will be raised immediately
  # and will terminate the whole application
  raise "oh no"
end
```

It can also be set to a single thread by accessing the attribute via the thread instance from inside or outside the thread:

```
t = Thread.new do
  Thread.current.abort_on_exception = true

  # The exception will be raised immediately
  # and will terminate the whole application
  raise "oh no"
end

# This could be done as well:
t.abort_on_exception = true
```

## Passing Data In and Out

Ruby offers a few ways to interact between the thread and its outer environment. The first way is by passing parameters to the new thread. This is done by passing the arguments to the new method and receiving them within the code like the regular Ruby block way, between two vertical bars:

```
Thread.new("Shay", "Friedman") do |first_name, last_name|
  puts "#{first_name} #{last_name}" # prints "Shay Friedman" to the screen
end
```

The second way to pass data to and from the thread is via three methods that are part of every Ruby thread: `[], []=`, and `key?`. The thread behaves like a hash and can contain symbols and values. The `key?` method checks whether a given symbol exists within the hash. These methods are accessible from within the thread and from outside it and make it easy to pass data around:

```
t = Thread.new do
  5.times do |ind|
    Thread.current[:current_index] = ind
    sleep 1
  end
end

10.times do
  sleep 0.5
  puts t[:current_index] if t.key?(:current_index)
end
```

The preceding code prints the following number sequence: 0, 1, 1, 2, 2, 3, 3, 4, 4, 4. It doesn't print every number two times because of the different intervals; instead, the `t` thread and the main thread are sleeping and waking up.

## Thread Priority

A thread priority can be set programmatically. This is useful in case you want your thread to be preferred over other threads.

This is done using the `priority` method. Table 8.1 describes the available priority values.

Setting the priority can be done from within the thread using the `Thread.current` method or from outside:

```
t = Thread.new do
  ... thread code ...
end

t.priority = 2
```

TABLE 8.1 Thread Priority Values

| Value       | Description   |
|-------------|---|
| -2 or lower | A thread that is scheduled after any other thread         |
| -1          | A thread that is scheduled after normal-priority threads  |
| 0           | Default priority  |
| 1           | A thread that is scheduled after highest-priority threads |
| 2 or higher | A thread that is scheduled before any other thread        |

The `pass` method can be used to generate a context switch. Therefore, if a high-priority thread is running and other threads should be able to run, too, the `pass` method should be called once in a while:

```
Thread.new do
  ... code ...
  Thread.pass
  ... code ...
end
```

## Thread State

A thread can be checked for its state, via the `status` method. The `status` method has several return values, each representing a different state of the thread. Table 8.2 describes the available states.

TABLE 8.2 Thread States

| status Value | Description   |
|--------------|---|
| "run"        | Thread is running or can be run.  |
| "sleep"      | Thread is sleeping.   |
| "aborting"   | Thread has been killed (via the <code>kill</code> method) but has not terminated yet. |
| False        | Thread has terminated.  |
| Nil          | Thread has terminated unexpectedly.   |

Another way to check whether a thread is alive is by using the `alive?` and `stop?` methods. A thread is considered alive when the thread hasn't terminated already. It is considered stopped when the thread is sleeping, is aborting, or has terminated.



```

t = Thread.new do
  sleep 5
end

t1 = Thread.new do
  raise "error!"
end

t.status # = "sleep"
t.alive? # = true
t.stop? # = true
t1.status # = nil
t1.alive? # = false
t1.stop? # = true

```

The thread state can also be altered. Ruby offers a set of methods that can pause, resume, and kill a thread:

- ▶ To pause a thread, the `Thread.stop` method should be used. This puts the thread to a “sleep” state. This can be called only from within the thread.
- ▶ To resume the thread, the `wakeup` or `run` methods can be used. `run` also signals the CPU, which means it has potential to start running more quickly than `wakeup`.
- ▶ To kill a thread, the `terminate` method should be used. When the method is used, the thread’s `ensure` clause is executed. If the `terminate!` method is used, the `ensure` clause is ignored, and the thread terminates immediately.

For example, the next sample runs a thread that writes numbers on the screen and pauses after each iteration. The user inserts a selection on the main thread, which either awakes the thread or terminates it (softly or not):

```

t = Thread.new do
  begin
    i = 0
    loop do
      puts i
      i = i + 1
      Thread.stop # pause the thread
    end
  ensure
    puts "Bye bye"
  end
end

choise = gets.chomp
while choise != 'q'
  case choise

```

```

    when 'r'
      t.run
    when 't'
      t.terminate
    when 't!'
      t.terminate!
    end

    choise = gets.chomp
end

```

## Thread Synchronization

When multiple threads work simultaneously, sometimes a problem arises that causes two or more threads to depend on each other. When this issue arises, a mechanism to synchronize between the threads is needed. Ruby supports a few kinds of synchronization techniques as part of the standard library thread.

### Mutex

The first synchronization mechanism is the `Mutex` class. `Mutex` is a locking mechanism that helps in situations when a critical section exists. To use it, before a thread accesses a critical section, it needs to lock the mutex. When the thread is done, the mutex should be unlocked. After the mutex has been locked, any other thread that tries to access it will be blocked until the mutex is unlocked.

The `Mutex` class has `lock` and `unlock` methods. However, the common scenario is done with the `synchronize` method. The `synchronize` method retrieves a code block and automatically locks the mutex while the code is running and unlocks it when it's over:

```

require 'thread' # Get the thread standard library

mutex = Mutex.new
t = Thread.new {
  mutex.synchronize {
    ... critical section code ...
  }
}
t1 = Thread.new {
  mutex.synchronize {
    ... critical section code ...
  }
}

```

As mentioned, `mutex lock` and `synchronize` are blocking for the other threads that access them. `Mutex` features two methods that will not block the thread if the mutex is locked: `try_lock` and `locked?`. They both return a Boolean value. `try_lock` returns a value that

indicates whether it succeeded in locking the mutex, and `locked?` returns a value that indicates whether the mutex is locked.

### Queues

If you have a producers-consumers design, you may sometimes need a way to interact between them. The communication can be done using mutexes, although the more appropriate solution is using a queue. The producers will add items to the queue, and the consumers will retrieve them.

Ruby supports two types of queues: `Queue` and `SizedQueue`. These are thread-safe FIFO queues. They support two main methods: `enq` and `deq` for enqueueing and dequeuing items. These queues also have a special behavior: If the `deq` method is called when the queue is empty, the call blocks the calling thread until an item is added to the queue. The difference between the queues is the limit on the number of items that can be defined on `SizedQueues`. Similar to `deq`, when a thread tries to enqueue an item and the sized queue is already full, the thread is blocked until the queue has the needed space. The maximum number of items can be defined on `SizeQueue`'s constructor or by the `max=` method.

For example, the next code sample writes the numerals 0 to 99 on the screen:

```
require 'thread'

# Defining a queue with 10 items at most
q = SizedQueue.new(10)

# The consumer thread
t = Thread.new do
  100.times { |i| puts q.deq }
end

# The producer thread
t1 = Thread.new do
  100.times { |i| q.enq(i) }
end

t.join; t1.join
```

### Monitor

Monitors are similar to mutexes. They enable synchronizing blocks using the `synchronize` method. However, there is one major difference between monitors and mutexes: A mutex can be synchronized using a mutex variable, and a monitor can be incorporated into an object.

For example, suppose we have the following class:

```
class ParallelDemo
  def run
    (1..100).each { |i| puts i; sleep(rand) }
  end
end
```

```

    end
end

```

The class has a method, `run`, which loops 99 times, writes each number, and sleeps for a random number of milliseconds.

Consider the next code:

```

p = ParallelDemo.new
t = Thread.new { p.run }
t1 = Thread.new { p.run }
t.join; t1.join

```

The output of this code would have been a random sequence of numbers rather than two 1 to 99 sequences that appear one after the other.

We can change the `ParallelDemo` class to add synchronization capabilities. This is done with the `monitor` standard library, which contains a `MonitorMixin` module and the `Monitor` class itself:

```

require "monitor"
class ParallelDemo
  include MonitorMixin
  def run
    (1..100).each { |i| puts i; sleep(rand) }
  end
end

```

Now we can change the execution code, as well, to synchronize run method invocations:

```

p = ParallelDemo.new
t = Thread.new { p.synchronize { p.run } }
t1 = Thread.new { p.synchronize { p.run } }
t.join; t1.join

```

Make sure that mutexes are closer to the OS than monitors and that their performance is better than the monitors' performance. In addition, you can use mutexes across several applications. Nevertheless, monitors are easy to use and fit a lot of cases where you need a quick and easy solution.

## Handling Files

Working with files is a common practice during application development. Ruby has various classes that help in reading and writing files, checking file properties, listing

directories, and running operations on files. The `File` class is the center of file operations in Ruby, and it contains most of the available file operations.

### SAMPLE FILE USED IN THIS SECTION

Code samples in this section use a `sample.txt` file, which contains the next text block:

```
This is the first line
This is the second line
```

---

## Reading Files

The `File` class has multiple methods that offer different ways for reading a file.

The first method group allows reading the file content directly into a variable. This is done using one of the methods: `read`, `readlines`, or `foreach`. `read` reads the whole file content into a variable, `readlines` returns the lines in the file as an array (a different delimiter can be defined), and `foreach` yields each line to its associated code block (a different delimiter can be defined):

```
content = File.read("Sample.txt")
puts content

# Output: "This is the first line
#         This is the second line"

lines = File.readlines("Sample.txt")
lines.each_with_index { |line, index| puts "#{index}: #{line}" }
# Output: " 1: This is the first line
#         2: This is the second line"

# Seperate the file by spaces:
File.foreach("Sample.txt", " ") { |c| print "_#{c}" }
# Output: "_This_is_the_first_line
#         This_is_the_second_line"
```

The second method group uses streams to read the file. To read a file as a stream, the stream should be opened first. This is done by the `File.open` method. There are two ways of calling it. The first one is to call it, store the returned `File` instance in a variable, use it, and then close it. The second way is to associate a code block to the call. This way the file stream will be closed when the code block ends. For example, the next two samples are equivalent:

```
file = File.open("Sample.txt")
puts file.read
file.close
```

```
File.open("Sample.txt") do |file|
  puts file.read
end
```

The `open` method receives another parameter that indicates the mode the file will be opened in. Table 8.3 describes the different available modes.

TABLE 8.3 File.open Modes

| Mode | Description  |
|------|--|
| "r"  | Read only. The default mode.   |
| "r+" | Read and write. Fails if file doesn't exist.                               |
| "w"  | Write only. Creates a new file or truncates the existing file.             |
| "w+" | Read and write. Creates a new file if file doesn't exist.                  |
| "a"  | Write only. Creates a new file or appends to the end of the existing file. |
| "a+" | Like "a" but allows reading as well.                                       |
| "b"  | Binary file mode. Should be added to one of the previous modes.            |

When the file stream is opened, several reading operations can be done: `read`, `readchar`, `getc`, `readline`, and `gets`.

`read`, without parameters, reads the entire file like we've seen previously in this section. When `read` is used inside a stream, it can be used to read the file by chunks—just pass the length of each read operation. This length is the maximum number of characters. If fewer characters exist, they will be returned:

```
file = File.open("Sample.txt")
while !file.eof?
  puts file.read(11)
end
file.close
```

```
#Output: "This is the
#         first line
#
#         This is th
#         e second li
#         ne"
```

`read` can also receive a buffer parameter that will contain the read buffer. This way the `read` call can replace the loop condition. The next code sample is identical to the one above:

```

file = File.open("Sample.txt")
buffer = String.new # The variable must be of type string
while file.read(11, buffer)
  puts buffer
end
file.close

```

`readchar` reads the file by characters. The char is returned as its ASCII code. `readchar` will raise an exception on end of file. The alternative to that is to use `getc`, which behaves the same but returns `nil` on the end of file.

`readline` reads the file line by line. Like `readchar`, an error is raised on end of file. Use `gets` as an alternative; it returns `nil` on end of file.

When a stream is used, a few `each` methods also become available: `each` and `each_byte`. `each` will yield to its associated code block every line in the file (a different separator can be defined). `each_byte` will iterate through the file bytes. Using the `each` methods saves the checks for the end of file:

```

file = File.open("Sample.txt")
file.each { |x| puts x }
file.close

```

## Writing Files

Writing to files in Ruby requires a writable stream. When such a stream is opened, several methods allow writing to the stream: `print`, `printf`, `puts`, and `write`.

`print` adds the given string to the stream. It can receive multiple arguments separated by commas, and it will convert them to strings (using `to_s`) and add them to the stream:

```

file = File.open("a.txt", "w")
file.print "One ", 1, " ", "Two ", 2
file.close
# a.txt now contains "One 1, Two 2"

```

### CHANGE PRINT SEPARATOR

`print` actually adds the given arguments to the stream and the global output record separator at the end. The default separator is `nil` by default but can be changed. The global output record separator can be accessed by the `$\` global variable:

```

$\ = ">>"

file = File.open("a.txt", "w")
file.print "One ", 1, " ", "Two ", 2
file.close

# a.txt now contains "One 1, Two 2>>"

```

`write` is very similar to `print`. The only difference between them is that `print` adds the global output record separator (\$\) after appending its arguments to the stream, whereas `write` doesn't do that.

`printf` is a way to append a formatted output to the stream. Look at the `printf` description in Chapter 5, "Ruby Basics," for more information.

`puts` acts like `print` except that it adds a new line at the end of each output. If `puts` is given an array, it prints every array item in its own line:

```
file = File.open("a.txt","w")
file.print ["one","two","three"], "Four"
file.close
# a.txt now contains "one
#                      two
#                      threeFour"
```

## Accessing File Properties

Sometimes you need to check or change file properties, such as the file size or whether it is read-only.

Ruby features methods to examine file properties:

```
file = "a.txt"

File.file?(file) # Is this a file?
File.directory?(file) # Is this a folder?
File.size(file) # The file size in bytes
File.zero?(file) # Is the file empty?

File.readable?(file) # Can the file be read?
File.writable?(file) # Can the file be written to?
File.executable?(file) # Can the file be executed?

File.mtime(file) # Last modification time
File.atime(file) # Last access time

file = File.join("D:", "Dev", "Scripts", "Sample.txt") # = "D:/Dev/Scripts/Sample.txt"
File.basename(file) # = "Sample.txt"
File.basename(file, "*") # The filename without any extension = "Sample"
File.basename(file, ".txt") # = The filename without .txt extension = "Sample"
File.basename(file, ".exe") # = The filename without .exe extension = "Sample.txt"
File.extname(file) # = ".txt"
File.dirname(file) # = "D:/Dev/Scripts"
File.split(file) # = ["D:/Dev/Scripts", "Sample.txt"]
```



You can also retrieve the file metadata into a single object by using the `File.stat` method:

```
stat = File.stat("a.txt")
stat.file?
stat.directory?
stat.readable?
stat.writable?
stat.executable?
stat.mtime
stat.atime
```

Changing file read-only state is done via the `File.chmod` method:

```
File.writable?("a.txt") # = true
File.chmod(0, "a.txt") # Make it read-only
File.writable?("a.txt") # = false
File.chmod(128, "a.txt") # Make it writable
File.writable?("a.txt") # = true
```

To update the modification and last access time, you use the `File.utime` method:

```
# Change a.txt modified date to 9/6/1983 and access time to 9/6/2009
File.utime(Time.local(2009, 9, 6), Time.local(1983,9, 6), "a.txt")
File.mtime("a.txt") # = 9/6/1983
File.atime("a.txt") # = 9/6/2009
```

## Listing Directories

You can list directories by using the `Dir` class, which holds directory-related methods.

To retrieve all files within a directory, the `entries` and `foreach` methods can be called.

`entries` returns the files as an array, and `foreach` yields each file to its associated code block:

```
arr = Dir.entries("IronRuby")
Dir.foreach("IronRuby") { |file| puts file }
```

Another way to retrieve the file list of a specific folder is by using `[]` or the `glob` methods. They are different from `entries` and `foreach` by their capability to filter the file list. `[]` returns the file list as an array, and `glob` yields each file to its associated code block. To define the directory on which the methods will be executed, `Dir.chdir` should be used; otherwise, the executed file directory is used:

```
Dir.chdir("D:/IronRuby")
# Get all rb files from D:\IronRuby
```

```
arr = Dir["*.rb"]
# Get rb files in all sub directories of D:\IronRuby
Dir.glob("**/*.rb") { |file| puts file }
```

## File Operations

The `File` class also features methods for modifying or deleting files:

```
# Rename a file:
File.rename("a.txt", "Example.txt")
# Delete a file:
File.delete("Example.txt")
# Create a directory
Dir.mkdir("SampleDir")
# Delete a directory
Dir.rmdir("SampleDir")
```

The standard library class `FileUtils` adds some more operations, such as copying and moving files, and improves existing operations:

```
require 'fileutils'

dir = "C:/Hello/Ruby/From/FileUtils"

# Create the whole directory tree
FileUtils.makedirs(dir)
# Copy a file from current directory to the new directory
FileUtils.copy("Sample.txt", dir)
# The copy method can copy multiple files as well:
FileUtils.copy(Dir.glob("*.rb"), dir)
# Move a single or multiple files from one directory to another
FileUtils.move(["a.rb", "b.rb"], dir)
# Delete a folder along with its content
FileUtils.remove_dir("C:/Hello")
# Delete a list of files and directories
#(directories are deleted recursively like remove_dir)
FileUtils.rm_r(["C:/temp/tempFiles", "C:/temp/temp.bin"])
```

These capabilities are available out-of-the-box with IronRuby installation. However, there are numerous more libraries that can add even more file handling capabilities to the language (for example, discovering the MIME type of a file). Look at the [RubyGems](#) section later in this chapter for information about how to locate, retrieve, and use Ruby libraries from the Internet.

## Reflection

Ruby is a very permissive language, as you've already seen time and time again. It allows developers to open any class, add new methods, remove others, include mixins, and more. Ruby isn't satisfied with just that, and so it provides some impressive reflection capabilities.

Reflection means that the code can investigate itself and its environment. Hence, you can retrieve a list of current defined variables, check what the superclass of an object is, and more. This gives you a lot of control over objects during the runtime of the application.

Reflection serves advanced needs of applications. For example, creating code dynamically or developing extremely generic classes. It can also help tremendously during debug sessions by making it possible to investigate the running objects.

## Finding Living Objects

The `ObjectSpace` module interacts with Ruby's Garbage Collector. This is why it can help us discover all living objects within our application.

`ObjectSpace` contains an `each_object` method that calls its associated block for every living object. A module or class type can be passed as a parameter to retrieve objects from this type only:

```
john = Human.new("John", "Doe")

ObjectSpace.each_object(Human) { |x| x.introduce }
# Output: "Hi, I'm John Doe"
```

### TYPES THAT OBJECTSPACE WON'T FIND

`ObjectSpace` will not return immediate types: `Fixnum`, `Symbol`, `true`, `false`, or `nil`.

For example:

```
num = 5 # This is a Fixnum
float = 5.5 # This is a Decimal
bignum = 9_999_999_999_999_999 # This is a Bignum

ObjectSpace.each_object(Numeric) { |x| puts x }

# Output:
# 9999999999999999
# 5.5
# 2.71828182845905
# 3.14159265358979
```

```
# 2.22044604925031e-016
# 1.79769313486232e+308
# 2.2250738585072e-308
```

Ensure the Fixnum variable named num doesn't appear on the list. The spare numbers are constants from the Math module.

---

## Investigating Objects

When we have an object, we can investigate its internals. We can get its methods, test whether the object responds to a specific method name, look at its inheritance hierarchy, and even access its internal variables.

Object class and class hierarchy can be retrieved in the following ways:

```
str = "Hello"

str.class # = String
str.class.superclass # = Object
str.class.ancestors # = [String, Enumerable, Comparable, Object, Kernel]
str.class.included_modules # = [Enumerable, Comparable, Kernel]
str.class.ancestors - str.class.included_modules # Class hierarchy only = [String,
Object]

# instance_of? returns true for the direct superclass
str.instance_of?(String) # = true
str.instance_of?(Object) # = false
str.instance_of?(Comparable) # = false

#kind_of? returns true for every superclass or mixin module:
str.kind_of?(String) # = true
str.kind_of?(Object) # = true
str.kind_of?(Comparable) # = true

# === is similar to kind_of?
String === str # = true
Object === str # = true
Comparable === str # = true
```

Method listing and information can be retrieved in the following ways:

```
str = "Hello"

str.methods # all public methods as an array
str.public_methods(false) # Public methods without inherited ones
str.public_methods(true) # Public methods with inherited ones
```

```

str.protected_methods(false) # Protected methods without inherited ones
str.private_methods(false) # Private methods without inherited ones
str.singleton_methods(false) # Singleton methods without inherited ones
str.class.methods # Class methods
String.methods # Same as str.class.methods

# Test if method exists as public or protected
str.respond_to?("chomp") # Does a method named "upcase" exist?
str.respond_to?(:chomp) # Works with symbols too
# Test if method exists as public, protected or private
str.class.public_method_defined?("chomp")
str.class.protected_method_defined?("chomp")
str.class.private_method_defined?("chomp")

```

Variables can also be listed:

```

john = Dentist.new("John", "Doe")

john.instance_variables # Get instance variables = ["@first_name", "@last_name"]
john.class.class_variables # Get class variables = nil
john.class.constants # Get class constants = nil
local_variables # get the current scope local variables = ["john"]

# Test if variable or const exist:
# Does @first_name exist as an instance variable?
john.instance_variable_defined?("@first_name")
# Does @@first_name exist as a class variable?
john.class.class_variable_defined?("@@first_name")
# Does FIRST_NAME const exist?
john.class.const_defined?(:FIRST_NAME)

```

## Invoke Methods and Set Variables Dynamically

The preceding section introduced ways to receive information about a specific object. This section shows you techniques to use these internal methods and variables.

To play with a class we know, let's construct a Demo class:

```

class Demo
  @@class_var = "I'm a class var!"
  def self.class_method
    puts @@class_var
  end
  def initialize(value)
    @instance_var = value
  end
end

```

```

private
def private_method(prefix)
  puts "#{prefix} #{@instance_var}"
end
end

```

Now that we have a class, let's see how we can invoke its methods, even the private ones:

```

# Invokes Demo class_method
Demo.send(:class_method) # Prints "I'm a class var!"
Demo.method("class_method").call

# Invoke Demo private instance method private_method with one parameter
d = Demo.new("Rocks!")
d.method(:private_method).call("IronRuby") # Prints "IronRuby Rocks!"
d.send("private_method", "IronRuby")

```

Invoking methods is not all we can do. Setting and getting variables is another capability that Ruby's reflection mechanism holds:

```

d = Demo.new("Rocks!")

# Get and set instance variables
d.instance_variable_get(:@instance_var) # = "Rocks!"
d.instance_variable_set(:@instance_var, "is DA BOMB!")

d.send(:private_method, "IronRuby") # Prints "IronRuby is DA BOMB!"

# Get and set class variables
Demo.send("class_variable_get", :@@class_var) # = "I'm a class var!"
Demo.send("class_variable_set", :@@class_var, "Oh yea!")

Demo.class_method # Prints "Oh yea!"

```

The `class_variable_get` and `class_variable_set` are private methods in Ruby 1.8; this is why we use the `send` method to invoke them.

Other than just setting variables, we can also get and set class constants:

```

Math::PI # = 3.1415...
Math.const_get(:PI) # = 3.1415...
Math.const_set(:PI, 3.2) # Set PIE constant to the high school PIE value
Math::PI # = 3.2

```

## SETTING CONSTANT VALUES

Setting a constant value via the `const_set` method shows a warning in case the interpreter is set to show warnings.

## Execute Code Dynamically

Maybe one of the strongest reflection capabilities in Ruby is the capability to execute code dynamically. This is done using the `eval` method. `eval` receives a string that contains a valid Ruby code and executes it, just as if it were written in the code file:

```
str = %q{puts "Hello!"}
eval str # Prints "Hello!"
```

`eval` runs in the current context. Two more `eval` methods can execute the code in a different context: `class_eval` (or its synonym, `module_eval`) and `instance_eval`. `class_eval` can execute code in the class context (`self` can be the class object), and `instance_eval` can execute code in the context of the class (`self` can be the class instance):

```
d = Demo.new("IronRuby")
d.instance_eval %q{puts "#{instance_var Rocks!}" } # Prints "IronRuby Rocks!"

Demo.class_eval "@@class_var = 'Changed by class_eval!'"
Demo.class_method # Prints "Changes by class_eval!"
```

## ACCESSORS IMPLEMENTATION

As you may recall, when we discussed accessors in Chapter 6, “Ruby’s Code-Containing Structures,” I mentioned that accessors were simply code generators. After we know what `eval` is, we can implement accessors ourselves. Let’s implement our own `attr_accessor`. First, I’ll open the `Module` class and add the accessor implementation:

```
class Module
  def unleashed_accessor(symb)
    code = <<CODE
    def #{symb}
      @#{symb}
    end

    def #{symb}=(value)
      @#{symb} = value
    end
  end
end

CODE
```

```

        # I use class_eval to define instance methods
        class_eval code
      end
    end
  end
end

```

---

After we have our accessor ready, we can use it:

```

class Sample
  unleashed_accessor :accessor
end

s = Sample.new
s.accessor = "Hello"
puts s.accessor # Prints "Hello" to the screen

```

## Marshaling

Most applications need to save data to a permanent location so that it is available every time the application runs. In Ruby, this is called *marshaling*. In other languages, it is sometimes called *serialization*.

Ruby has two techniques for marshaling: binary and textual. Both of these techniques have their advantages and disadvantages—binary marshaling is generally faster and takes less space whereas textual marshaling ends up as a human readable file that can be modified outside of the application context.

### Binary Marshaling

There are two methods for binary marshaling: `Marshal.dump` and `Marshal.load`. As you can guess from their names, `Marshal.dump` serializes the object to a binary format, and `Marshal.load` constructs the object from the binary stream:

```

john = Human.new("John", "Doe")

# Save john to a file
f = File.open("john", "w+")
Marshal.dump(john, f)
f.close

# Load john from the file
File.open("john", "r") do |f|
  john = Marshal.load(f)
end

john.introduce # Prints "Hi, I'm John Doe"

```



## Textual Marshaling

Binary marshaling has two disadvantages. First, when the object structure is changed, the binary format changes significantly. Second, the formats are not readable by humans.

This is what YAML is for. YAML is a standard library that helps in serializing objects to text. After you have the text saved on a file, for example, there is no problem with changing the values, and the new ones will be reflected when the object is deserialized.

YAML, just like the Marshal module, has dump and load methods:

```
require 'yaml'
john = Human.new("John", "Doe")

# Save john as text
text = YAML.dump(john)
# Load john from the text
john = YAML.load(text)

john.introduce # Prints "Hi, I'm John Doe"
```

The text that the variable `john` was converted to looked like the following:

```
--- !ruby/object:Human
first_name: John
last_name: Doe
```

If we modify the code to replace one of the values in the string, we will see them change after we load the object:

```
require 'yaml'
john = Human.new("John", "Doe")

# Save john as text
text = YAML.dump(john)
# Replace "John" with "Joanne"
text["John"] = "Joanne"
# Load john from the text
john = YAML.load(text)

john.introduce # Prints "Hi, I'm Joanne Doe"
```

## RubyGems

RubyGems is a centralized place for Ruby libraries. It is similar to .NET's Global Assembly Cache (GAC). RubyGems allows installing and uninstalling Ruby libraries, signing libraries, listing the available libraries, and version management.

RubyGems is not an integral part of Ruby 1.8 or IronRuby 1.0. It is included in the installation package as an external utility.

### Installing RubyGems

In IronRuby's installation directory, you will find the RubyGem utility, called `igem`. RubyGems is ready for use.

Calling `igem help` can show you a short help document about RubyGems. If you want to get more information about a specific command, run `igem help <COMMAND>`.

### Installing Gems

RubyGems is all about the gems, right? So the next thing we need to do is to find gems and install them. The list of available gems is available via the `igem` utility. `igem list -l` shows the list of local gems, and `igem list -r` shows the list of gems on the gems server. To get a brief description of the gem, add `--details` to the end of the command: `igem list -r --details`.

After we've decided which library to install, we can use the `igem install <GEM>` command. We can install the `simple-rss` library, which features a simple way to read RSS and ATOM feeds:

```
> igem install simple-rss
Successfully installed simple-rss-1.2
1 gem installed
Installing ri documentation for simple-rss-1.2...
Installing RDoc documentation for simple-rss-1.2...
```

### Using Installed Gems

After a gem is installed, you can use it within your application. To use gems, we need to first require RubyGems and then require our gem. In our case, we write the next couple of lines:

```
require 'rubygems'
require 'simple-rss'
```

Now we can use the `simple-rss` library and read our favorite RSS feed. We use the `open-uri` library, as well, which is a part of the standard library. This library helps in reading HTTP, HTTPS, and FTP streams.

The entire application, which reads my personal blog RSS feed and prints general details to the screen, looks like this:

```
require 'rubygems'
require 'simple-rss'
require 'open-uri'

rss = SimpleRSS.parse open('http://feeds.feedburner.com/ShayTalksAbout')

puts rss.feed.title # = "IronShay"
puts rss.feed.link # = "http://www.IronShay.com"
puts rss.items.size # = 15 (number of items in the RSS feed)

# Print post titles
rss.items.each { |post| puts post.title }
```

## Rake

Rake is a gem library that is used to build and install ruby applications. Rake is built of *rakefiles*. A rakefile is a file written in Ruby code that supplies the instructions to the build process. Rake is similar to *make* or *ant*.

Rake enables you to create a task base flow, which builds and configures the application and its environment. For example, we can define a task that creates folders for the application, build code, run tests, configure a database, and eventually send an email to the administrator that the process is complete.

A simple rakefile looks like the following (the file is actually called `rakefile`):

```
task :create_directories do
  puts "Creating folders..."
  #...creating folders...
end

# "Copy files" is dependent on "Create Directories"
task :copy_files => :create_directories do
  puts "Copying files..."
  #...copy files
end
```

To run this rakefile, we use the `irake` tool, which is in the IronRuby installation folder. `irake` gets as a parameter the name of the task to start with. It then looks for a file named

rakefile and executes the code inside the given task and its dependencies (the dependencies will be executed before):

```
> irake copy_files  
(in C:/IronRubyApps)  
Creting folders...  
Copying files...
```

## RAKE IS NOT JUST FOR RUBY APPLICATIONS

Rake is a build tool, and it is not restricted to building Ruby files. The build tasks are written in Ruby, and they can be used to execute building processes of applications written in other programming languages. For instance, building .NET code can be done by running the msbuild process via IronRuby.

Delving deeper into rake is beyond the scope of this book. For further information about the rake tool, visit <http://rake.rubyforge.org>.

## IronRuby RubyGems Limitations and Expertise

In version 1.0 of IronRuby, there are a few known limitations apply to using gems:

- ▶ Native gems are not supported. Native gems are gems that include native code (C code) extensions.
- ▶ Signed gems are not supported.

However, IronRuby has an expertise that no other Ruby implementation offers: the ability to use CLR extensions. An extension can be written in one of the .NET languages and still be used from Ruby.

Chapter 19, “Extending IronRuby” covers how to write CLR extensions.

## Finding Gems

There are numerous websites that catalog gems. The big sites contain thousands of gems that implement almost anything you might need.

The big gem catalog sites are

- ▶ GitHub: <http://www.github.com>
- ▶ Gemcutter: <http://gemcutter.org>

## Design Patterns

In the world of programming, chaos is common. Applications that get bigger by the minute lose their initial clean and smooth way of coding and become one massive patches container. Design patterns are planned to help avoid this.

Design patterns should not be followed blindly. In essence, they are just templates, and they should be shaped to fit the exact needs of every application.

The following are some common design patterns that are easy to implement in Ruby.

### The Strategy Pattern

The strategy pattern is used to change the way an object works by passing it a strategy object. For example, in a basketball match, the team has different defensive strategies. If we need to code a basketball game, we can use the strategy pattern, for example:

```
class BasketballTeam
  def initialize
    @players = []
  end

  def play_defense(strategy)
    strategy.play(@players)
  end
end

class ZoneDefense
  def play(players)
    # ... do zone defense ...
  end
end

class ManToManDefense
  def play(players)
    # ... do man-to-man defense ...
  end
end

team = BasketballTeam.new

# Play zone defense
team.play_defense(ZoneDefense.new)

# Or play man-to-man defense
team.play_defense(ManToManDefense.new)
```

As you can see from this code sample, the `BasketballTeam` class does not have to know its strategy objects. It just needs to know its interface to use them. Now we can add multiple other defensive strategies without modifying the main class, `BasketballTeam`.

Ruby's duck typing capability makes it easy to create strategy objects. We do need to declare an interface or an abstract class. In our basketball sample, all a class needs to become a valid strategy class is to contain a `play` method.

The code is currently "unsafe" because there is no check of the strategy object before using it. To make it safer, a validation should be added to check whether the object responds to the `play` method:

```
def play_defense(strategy)
  strategy.play(@players) if strategy.respond_to?(:play)
end
```

After this change, if the strategy object isn't valid, the team will not play defense (and will probably lose the game).

Even though it might seem like a good idea to use `respond_to?` all the time, it is not used that much in practice. The more common way of dealing with that is by unit testing the code.

The strategy pattern can be written in another way that takes advantage of another Ruby feature, blocks. Instead of passing a strategy object, we can provide a strategy code block:

```
class BasketballTeam
  def initialize
    @players = []
  end

  def play_defense(&strategy)
    strategy.call(@players)
  end
end

team = BasketballTeam.new

# Play zone defense
team.play_defense do |players|
  # ... do zone defense ...
end

# Or play man-to-man defense
team.play_defense do |players|
  # ... do man-to-man defense ...
end
```

**Real-World Samples**

It's impossible to list all possible uses for the strategy pattern here, but it is possible to categorize a couple common uses for the strategy pattern:

- ▶ **Storage access:** Different strategy class for different storage locations (memory, database, file system, and so on)
- ▶ **Formatting:** Different strategies for every format (text, Word document, OpenOffice document)

**The Iterator Pattern**

The iterator pattern is used to generate a way of accessing an object's inner collection of subobjects. An iterator doesn't necessarily have to loop over a list of objects. It is just a forward-only cursor-based scan of the object's inner collection.

There are two kinds of iterators: internal iterators and external iterators. Internal iterators iterate through the collection and pass the current item to the users on each iteration. External iterators expose an iteration object (sometimes called a cursor), and the users advance the iteration and check its state by themselves.

For example, in our `BasketballTeam` class, we can add an iterator to go through the players of the team.

Let's first create a `BasketballPlayer` class that will hold player information:

```
class BasketballPlayer
  attr_accessor :first_name, :last_name, :height

  def initialize(first_name, last_name, height)
    @first_name = first_name
    @last_name = last_name
    @height = height
  end
end
```

Next we'll implement methods to add players to a team:

```
class BasketballTeam
  def initialize
    @players = []
  end

  def add_player(player)
    @players << player
  end
end
```

This section is about the iterator pattern, so let's implement an iterator method. We will call it `each`, which is the convention for iterators in Ruby:

```
def each
  @players.each { |player| yield player }
end
```

This wraps up the iterator pattern implementation. We can now iterate through the basketball team members with ease, as follows:

```
team = BasketballTeam.new
team.add_player BasketballPlayer.new("Shaquille", "O'Neal", 7.1)
team.add_player BasketballPlayer.new("James", "LeBron", 6.8)

team.each { |player| puts player.first_name }
# Writes "Shaquille" and "James" to the screen
```

After we have this defined, we can add lots more functionality to our class. This can be done using the `Enumerable` mixin, which was discussed in Chapter 6. All we have to do is to implement `<=>` on the iterator objects (`BasketballPlayer`) and include `Enumerable` in our `BasketballTeam` class:

```
class BasketballPlayer
  def <=>(other)
    height <=> other.height
  end
end

class BasketballTeam
  include Enumerable
end
```

Now we can sort the team members, check for membership, and more:

```
lakers = BasketballTeam.new
lakers.add_player BasketballPlayer.new("Andrew", "Bynum", 7.0)
lakers.add_player BasketballPlayer.new("Pau", "Gasol", 7.0)
lakers.add_player BasketballPlayer.new("Trevor", "Ariza", 6.8)
lakers.add_player BasketballPlayer.new("Kobe", "Bryant", 6.6)
lakers.add_player BasketballPlayer.new("Derek", "Fisher", 6.1)

# Sort players by height
lakers.sort # = [Fisher, Bryant, Arize, Gasol, Baynum]

# Check whether there are any players below 6ft
lakers.any? { |player| player.height < 6 } # = false
```



**Real-World Samples**

The iterator pattern is one of the most commonly used patterns. It appears in data collections, streams (like files), strings, and more.

**The Command Pattern**

The command pattern aims to separate our operation code from the rest. It should support executing a command and undoing it, too. The idea is to have different command classes that support a single interface (do and undo). After a command is executed, it is added to a command container. When an undo request comes in, the last command from the list should be undone and removed from the list.

For example, let's add a mechanism that can enable the user to directly add new players to our basketball team:

```
class BasketballTeam
  def show_menu
    puts "Add New Players"
    puts "Q to exit or the player info: first name,last name,height"
    gets.chomp
  end

  def get_new_users
    input = show_menu
    while input.downcase != "q"
      fn, ln, h = input.split(',')
      add_player BasketballPlayer.new(fn, ln, h.to_f)
      input = show_menu
    end
  end
end
```

Now we can incorporate the command pattern here and make the flow clearer and even allow undoing!

First, I'll change the BasketballTeam class to support commands:

```
class BasketballTeam
  attr_accessor :players

  def initialize
    @players = []
    @commands = []
  end

  def add_player(player)
    command = AddBasketBallPlayerCommand.new(self, player)
```

```

    command.execute

    @commands << command
  end

  def undo
    return if @commands.empty?

    @commands.pop.undo
  end
end

```

The preceding code uses the `AddNewPlayerCommand` class. Let's code it:

```

class AddBasketBallPlayerCommand
  def initialize(team, player)
    @team = team
    @player = player
  end

  def execute
    @team.players << @player
  end

  def undo
    @team.players.delete @player
  end
end

```

After these changes, the command class takes full control over the player addition process, and the `BasketballTeam` class is used as the container of the commands. On every command execution, the command object is inserted into a list of commands and into every undo request. The last command is removed from the list while undoing its operation.

The latest changes make it possible to write the next code lines:

```

lakers = BasketballTeam.new
lakers.add_player BasketballPlayer.new("Kobe", "Bryant", 6.6)
lakers.add_player BasketballPlayer.new("Derek", "Fisher", 6.1)
lakers.undo

lakers.players # = [Bryant]

```

We can also take advantage of procs (another Ruby feature) and make this pattern more “developer friendly.”

To do that, we'll create a single command class that will hold two procs, one for `execute` and one for `undo`:

```
class Command
  def initialize(execute, undo)
    @execute = execute
    @undo = undo
  end

  def execute
    @execute.call
  end

  def undo
    @undo.call
  end
end
```

This class can save us the need for a special command class for every command. When we have this class defined, we can delete the `AddNewPlayerCommand` class and replace the `add_player` method code with the following:

```
def add_player(player)
  command = Command.new(Proc.new { @players << player },
                        Proc.new { @players.delete(player) })

  command.execute

  @commands << command
end
```

### Real-World Samples

The command pattern is widely used in applications that support multiple undo/redo operations (for example, text and image editors). It is also used when rollback support is needed (to undo an entire operation when the operation fails).

The command pattern comes in various shapes and has several uses. It is good practice to use it when you have commands to execute that can be separated logically into their own command classes.

## The Singleton Pattern

The singleton pattern is one of the most popular patterns. It is used in almost every application, probably as a result of its simplicity and added value.

The singleton pattern is described as a class that can only have a single instance, and this instance is globally accessible. To make our `BasketballTeam` class a singleton class, we have to add a class method that will return an instance of the class:

```
class BasketballTeam
  @@instance = BasketballTeam.new
  def self.instance
    @@instance
  end
end
```

The pattern also says that the class can have only one instance. To enforce that, we should make `BasketballTeam.new` unavailable. We can do so by making the `new` method private:

```
class BasketballTeam
  private_class_method :new
end
```

With these code lines added to the class, we can now use it as follows:

```
BasketballTeam.instance.add_player BasketballPlayer.new("Kobe", "Bryant", 6.6)
BasketballTeam.instance.players # = [Bryant]

lakers = BasketballTeam.new # Error!
```

In Ruby, all the code we've written is unneeded. Ruby has it ready for us already via the `Singleton` mixin module. All we have to do to convert a class to a singleton class is to include the `Singleton` module.

If we had not converted the `BasketballTeam` class to a singleton class, this is what we would have done:

```
require 'singleton'

class BasketballTeam
  include Singleton
end
```

After we do so, the `BasketballTeam` is a singleton class that can be accessed through the `instance` method and cannot be initialized using the `new` method.

**Real-World Samples**

As mentioned previously, the singleton pattern is one of the most commonly used patterns. It is mostly used when shared resources are at hand, as in the following:

- ▶ Logger classes.
- ▶ Settings classes. The settings are loaded once and are accessible through accessors.

**The Observer Pattern**

The observer pattern, as its name implies, is used when several objects are interested in changes made to another object. For example, when a player scores during a basketball match, the scoreboard and the statistics board should be updated.

We can solve this problem in various different ways. Most of them will involve coupling between classes, timers, or threads. The more sane way is to use the observer pattern. To do that, let's update our application first:

```
class Scoreboard
  def update(team, score)
    #... update scoreboard ...
  end
end

class StatisticsBoard
  def update(team, score)
    #... update stats board ...
  end
end

class BasketballTeam
  def score(amount)
    #... update score tables ...
  end
end
```

Now our application contains the needed classes and methods. The task now is to make the score method of the `BasketballTeam` class notify the `Scoreboard` and `StatisticsBoard` classes about the change. For that, I will add an `observers` collection to the `BasketballTeam` class and methods to add and remove observers. When this is done, I can use the `update` method to notify the added observers of changes:

```
class BasketballTeam
  def initialize
    @observers = []
  end
```

```
def add_observer(observer)
  @observers << observer
end

def remove_observer(observer)
  @observers.delete(observer)
end

def score(amount)
  @observers.each { |o| o.update(self, amount) }
end
end
```

After this code change, we can write the following code to notify the Scoreboard and StatisticsBoard classes of every score update:

```
team1 = BasketballTeam.new
scores = Scoreboard.new
stats = StatisticsBoard.new
team1.add_observer(scores)
team1.add_observer(stats)

team1.score(2) # Invokes the scores.update and stats.update methods
```

Ruby is distributed with a standard library that adds the observer pattern functionality with only a couple of lines. The library is called *observer*. To use it, we have to require the library and then include the Observable module within our target class. For example, our BasketballTeam class could have been rewritten using the observer library:

```
require 'observer'
class BasketballTeam
  include Observable

  def score(amount)
    changed
    notify_observers(self, amount)
  end
end
```

This code achieves exactly the same updates as the earlier examples.

Note that when using the observer library, before notifying the observers, the `changed` method should be called. If the `changed` method sets a Boolean value to true (and only in this case), the `notify_observers` method will then really notify the observers. After the `notify_observers` method notifies the observers, it sets the Boolean value back to false.

**Real-World Samples**

The observer pattern is used in many cases, including the following:

- ▶ User interface objects that notify of changes in the UI (a click on a button, for instance)
- ▶ Database objects that notify of changes in the data
- ▶ Parallel tasks that notify of their progress

In .NET languages, the observer pattern is rarely used this way. You might be more familiar with it as used for events and delegates.

**The Builder Pattern**

Building a basketball team isn't an easy task. Adding players is just one of many different tasks. The team should also have a coach, a general manager, a court, a gym, and so forth. The builder pattern simplifies such complicated building processes. The pattern instructs to create a class that builds another class. That way, the building process is easier and more structured.

Let's start by adding more information about the basketball team in the `BasketballTeam` class:

```
class BasketballTeam
  attr_accessor :players, :coach, :general_manager
  attr_accessor :courts

  def initialize
    @players = []
    @courts = []
  end

  def add_player(player)
    @players << player
  end

  def add_court(court)
    @courts << court
  end
end
```

Now I want to add two more classes: `Human` and `Court`. Two more classes will inherit from the `Court` class: `TrainingCourt` and `GameCourt`. The `Human` class will hold nonplayer team members (like the coach), and the `Court` classes will be added to the `courts` collection and hold information about the courts:

```
class Human
  attr_accessor :first_name, :last_name

  def initialize(first_name, last_name)
    @first_name = first_name
    @last_name = last_name
  end
end

class Court
  attr_accessor :name

  def initialize(name)
    @name = name
  end
end

class TrainingCourt < Court; end;
class GameCourt < Court; end;
```

With all of these objects, which are necessary for our BasketballTeam class, it gets pretty complicated to build a basketball team. The developer needs to be familiar with the different classes and know what class to use for the different attributes. We spare the developer the headaches by featuring a builder class that handles this process and exposes the built BasketballTeam object:

```
class BasketballTeamBuilder
  def initialize
    @team = BasketballTeam.new
  end

  def team
    @team
  end

  def add_player(first_name, last_name, height)
    @team.add_player BasketballPlayer.new(first_name, last_name, height)
  end

  def add_court(name, training = false)
    @team.add_court training ? TrainingCourt.new(name) : GameCourt.new(name)
  end
end
```



```

def set_coach(first_name, last_name)
  @team.coach = Human.new(first_name, last_name)
end

def set_general_manager(first_name, last_name)
  @team.general_manager = Human.new(first_name, last_name)
end
end

```

When the builder class exists, we can build a basketball team with ease:

```

builder = BasketballTeamBuilder.new
builder.set_coach("Phil", "Jackson")
builder.set_general_manager("Mitch", "Kupchak")
builder.add_court("Stapeless Center")
builder.add_court("Stapeless Center", true)
builder.add_player("Andrew", "Bynum", 7)
builder.add_player("Pau", "Gasol", 7)
builder.add_player("Trevor", "Ariza", 6.8)
builder.add_player("Kobe", "Bryant", 6.6)
builder.add_player("Derek", "Fisher", 6.1)

lakers = builder.team

```

Ruby allows us to improve the builder class and make it more developer friendly. This is done with the `method_missing` method (see Chapter 2, “Introduction to the .NET Framework”).

For example, we can improve the way of adding courts. In the preceding sample, I add the same court as a training and a game court. By using the `method_missing` method, I can join these code lines to a single one and make it more readable.

My `method_missing` implementation will parse the requested method name and add a training/game court. Thus, I can use methods such as `add_game_court` or even `add_training_and_game_court`:

```

class BasketballTeamBuilder
  def method_missing(name, *args, &block)
    # Use default implemenetation if the method name
    # doesn't match our exepectations
    super(name, *args, &block) if name.to_s !~ /(add_)([a-z_]+)(_court)/

    # Remove the beginning add_ and ending _court
    # and replace word delimiter with a space
    add_what = name.to_s.gsub("add_", "").gsub("_court", "").gsub("_", " ")

```

```
add_what.split(" and ").each do |val|
  # Decide whether the user wants to add a training court
  training = (val == "training")
  # Add the court using the name passed as a parameter
  add_court(args[0], training)
end
end
end
```

The preceding code sample enables us to minimize the next two lines

```
builder.add_court("Stapeless Center")
builder.add_court("Stapeless Center", true)
```

into a single one:

```
builder.add_training_and_game_court("Stapeless Center")
```

A side effect of this is making the code more readable for other developers.

### Real-World Samples

The builder pattern is used widely when the process of building a class is complex, including in the following scenarios:

- ▶ Generating documents (XML, HTML, Word, OpenOffice, and so on)
- ▶ Generating a user interface
- ▶ Defining unit tests

## Domain-Specific Languages

Domain-specific languages (DSLs) are languages dedicated to solving a specific problem. DSLs can be a part of a bigger language (and “bend” it to its needs) or an entirely different language. One of the most popular DSLs is the regular expressions language. Its syntax is very different from any other programming language and is specific to the problem of parsing text.

In Ruby, creating DSLs is easy. The language is permissive by nature and gives us various options to “bend” it to our needs. Because we can skip parentheses, for example, our methods can be read as language keywords, which makes the method feel like an integrated part of the language.

Let’s create our own DSL. I want to create a language that can be used to interpret people’s desires and demands.

The end result can enable us to write the following code lines:

```
I.want "a big house"
I.want 5.dogs
I.demand 1_000_000.bucks
I.demand "a fast car"

My.desires # Prints "a big house, 5 dogs"
My.demands # Prints "1000000 bucks, a fast car"
```

Creating this DSL is pretty straightforward. If you look carefully, you can separate the classes and methods.

We have two classes in our DSL: `I` and `My`. The `I` class has two class methods: `want` and `demand`. The `My` class has two class methods, as well: `desires` and `demands`.

The `I` class actually has two arrays: `@@desires` and `@@demands`. These arrays store the desires and demands written in the code:

```
class I
  @@desires = []
  @@demands = []

  def self.want(desire)
    @@desires << desire
  end

  def self.demand(dem)
    @@demands << dem
  end
end
```

The `My` class inherits from `I` and adds the `desires` and `demands` methods:

```
class My < I
  def self.desires
    puts @@desires
  end

  def self.demands
    puts @@demands
  end
end
```

This is enough to make I.want "a big house" available. I want to make I.want 100000.dollars a valid syntax, too.

To achieve that, I first add a HumanWish class that will hold a single desire or demand:

```
class HumanWish
  def initialize(wish, amount=nil)
    @wish = wish
    @amount = amount
  end

  def to_s
    if @amount != nil
      "@amount #@wish"
    else
      @wish
    end
  end
end
```

I have also added a to\_s implementation to ease the task of printing it out. Now that we have this defined, let's update the I class to add HumanWish classes to the @@desires and @@demands arrays rather than simple strings:

```
class I
  @@desires = []
  @@demands = []

  def self.want(desire)
    if desire.is_a? String
      desire = HumanWish.new(desire)
    end

    @@desires << desire
  end

  def self.demand(dem)
    if dem.is_a? String
      dem = HumanWish.new(dem)
    end

    @@demands << dem
  end
end
```

Notice that before adding the demand or desire to its corresponding array, I validate it is not a simple string. In that case, I convert the string to a `HumanWish` object before adding it to the array.

After the infrastructure is ready, we should make sure that `1000000.dollars` returns a `HumanWish` object. To do that, I will open the `Numeric` class and add a `method_missing` implementation:

```
class Numeric
  def method_missing(name)
    HumanWish.new(name, self)
  end
end
```

The preceding code means that you can write whatever you want along with a numeric value and it will generate a `HumanWish` out of it:

```
5.houses.is_a? HumanWish # = true
puts 5.houses # Prints "5 houses" (because of the HumanWish.to_s implementation)
```

Now the DSL is complete, the code I've written at the beginning of the section runs, and you can also write and store your desires and demands:

```
I.want "a big house"
I.want 5.dogs
I.demand 1_000_000.bucks
I.demand "a fast car"

My.desires # Prints "a big house, 5 dogs"
My.demands # Prints "1000000 bucks, a fast car"
```

### Real-World Samples

DSLs are often used without people realizing they are actually DSLs, as in the following:

- ▶ Regular expressions
- ▶ XPath
- ▶ Ruby's unit testing framework
- ▶ Rake and ant

## Summary

This chapter covered some of Ruby's advanced topics. You learned about threads and writing parallel code, handling files using the different file classes, taking advantage of Ruby's powerful metaprogramming capabilities, marshaling objects to and from files, using Ruby libraries from the web via RubyGems, and ways to implement several design patterns in the Ruby language.

Familiarity with these concepts can help you design more scalable and robust applications (and applications that are more “change friendly” for when requirements change)—a necessity for big applications.

*This page intentionally left blank*

# PART III

## IronRuby Fundamentals

### IN THIS PART

- CHAPTER 9 .NET Interoperability Fundamentals 207
- CHAPTER 10 Object-Oriented .NET in IronRuby 239



*This page intentionally left blank*

## CHAPTER 9

# .NET Interoperability Fundamentals

The previous chapters covered the Ruby language, from basics to advanced topics. Now it's time to look at IronRuby's specialty: its interoperability with the .NET Framework. It can speak with .NET classes and frameworks as if they were natural Ruby objects.

In this chapter, you learn the fundamentals of the interoperability between .NET and IronRuby, such as the transition of .NET objects as they become part of the Ruby language, enhancements that IronRuby adds to Ruby, and more.

## Bringing .NET into Ruby

The interoperability of IronRuby with the .NET Framework is a built-in support. You can start talking with .NET objects right away. The `System` namespace is also available without any direction. However, other .NET assemblies are not included directly and should be loaded before using.

Three methods can load .NET assemblies, all of them are part of the global `Kernel` class: `load`, `load_assembly`, and `require`. Each method has its own behavior, and it's important to know the different capabilities of each.

### **require**

`require` loads the given assembly only once and returns `true`. If it is called again with the same assembly, the call is ignored and the method returns `false`.

It can retrieve a string as a parameter that represents the library to load. Regarding .NET assemblies, there are various ways to pass them to the method.

## IN THIS CHAPTER

- ▶ Bringing .NET into Ruby
- ▶ .NET Code Mapping
- ▶ Using .NET Objects
- ▶ Special IronRuby Methods
- ▶ CLR Objects and Ruby's Reflection
- ▶ The Basic Object

**By Filename**

The filename can be passed with or without its extension. IronRuby looks for the assembly with three extensions (.dll, .so, and .exe). The search is done in the folders from the \$LOAD\_PATH variable:

```
require "CustomAssembly"
require "CustomAssembly.dll"
```

**By Full Path**

The given full path can contain the file extension or not. If not, IronRuby looks for the file with .dll, .so, and .exe extensions. Note that the path separator is / or \\:

```
require "d:/MyAssemblies/CustomAssembly"
require "d:\\MyAssemblies\\CustomAssembly.dll"
```

Another way to write a path is by using a single-quoted string. If you write a path in this way, you don't have to escape the backslashes:

```
require 'd:\\MyAssemblies\\CustomAssembly' By Home Path
```

The home path is the path in the HOME environment variable. The given assembly name (with or without extension) is searched in the Env["HOME"] directory (not in its subdirectories). To sign the interpreter to load the assembly from the home directory, the assembly path should start with ~/ or ~\\:

```
require "~/CustomAssembly.dll"
require "~\\CustomAssembly"
```

**By Strong Name**

When a strong name is used, the assembly is loaded using the System.Reflection.Assembly.Load method, which looks for the assembly in the Global Assembly Cache (GAC), in the installation directory, and in all the directories that have been used to load assemblies in the current session:

```
require "CustomAssembly, Version=1.0.0.0,
      Culture=neutral,
      PublicKeyToken=fef93daee49d7d60"
```

If the assembly is not signed, the PublicKeyToken can be written as "PublicKeyToken=null".

**REQUIRING .NET FRAMEWORK ASSEMBLIES**

When you need .NET Framework assemblies, like System.Data or System.Windows.Forms, there is no difference than in other custom assemblies. You need to provide a strong name or use a specific path using one of the possible inputs.

However, IronRuby contains within its Lib folder (under the IronRuby installation folder) some of the commonly used assemblies. The folder contains Ruby files that have one line in them: a `require` statement with the assembly strong name. The Lib folder is one of the folders in the `$LOAD_PATH` variable, so requiring the Ruby files there is direct, without the need to specify the full path.

The available assemblies in the Lib folder are `System`, `System.Data`, `System.Drawing`, `System.Windows.Forms`, `PresentationCore`, `PresentationFramework`, and `WindowsBase`.

As a result, instead of writing

```
require "System, Version=2.0.0.0, Culture=neutral,
      PublicKeyToken=b77a5c561934e089,
      processorArchitecture=MSIL"
```

It is possible to write the following:

```
require "System"
```

---

## load\_assembly

`load_assembly` is a special IronRuby method that is intended to make it easier to distinguish between .NET assemblies loading and other Ruby files.

`load_assembly`, unlike `require`, loads the assembly every time it is called.

The method receives a full or partial assembly name. The assembly is looked for in the GAC, in the installation folder, and in the loaded assemblies directories cache:

```
load_assembly "System.Management"
load_assembly "System.Management, Version=2.0.0.0,
      Culture=neutral,
      PublicKeyToken=b03f5f7f11d50a3a"
```

`load_assembly` has another optional parameter. This parameter contains the initializer class name. The initializer class is used for IronRuby libraries that are implemented in other .NET languages (like a part of the standard library). It allows these classes to map Ruby objects to .NET ones and initialize themselves in the Ruby context. For more information about using initializers, see Chapter 19, “Extending IronRuby,” where it is demonstrated as part of the discussion about creating IronRuby libraries using .NET languages.

The major difference between `load_assembly` and `require` is the capability of `load_assembly` to load assemblies from the GAC with their partial name. This is not available via `require` (only using the strong name):

```
require "System.Management" # Error!
load_assembly "System.Management" # OK
```

## load

`load` is identical to `require` with a few exceptions:

- ▶ `load` loads the assembly every time it is called.
- ▶ For full paths, `load` requires the path to contain the expected extension.

## The \$LOAD\_PATH Variable

`$LOAD_PATH` is a global variable that contains an array of locations that the `load` and `require` methods are looking at to find requested assemblies.

The `$LOAD_PATH` variable is set upon IronRuby startup. Its source is the `ir.exe.config` file, which exists within the IronRuby installation dir, right next to the `ir.exe` file.

Inside this XML file, under the `options` element, there are `set` elements. The one that has "LibraryPaths" as its option attribute value is `$LOAD_PATH` related. Its `value` element contains the assembly search directory paths, delimited by a semicolon. If you have a certain directory where you store your Ruby libraries or .NET assemblies, you can add its path to the semicolon-separated list. Doing so adds the directory to every current and future IronRuby application on the machine.

For a more local change, the `$LOAD_PATH` variable can be modified during runtime, which affects only the current running IronRuby instance. To add another path to the assembly lookup list during runtime, just add it to the `$LOAD_PATH` array:

```
require "CustomAssembly.dll" # Error!
$LOAD_PATH << "C:/CustomAssemblies"
require "CustomAssembly.dll" # OK
```

## .NET Code Mapping

The major languages of the .NET Framework, C# and VB.Net, have similarities and differences to Ruby. Classes appear in all of them, and Ruby's modules are similar to .NET's namespaces (not in the mixin meaning, though). Interfaces, on the other hand, do not exist in Ruby at all. To solve this collision, IronRuby maps .NET objects to similar Ruby objects. Table 9.1 lists the mapping.

TABLE 9.1 .NET to Ruby Object Mapping

| .NET Object | Ruby Object |
|-------------|-------------|
| Namespace   | Module      |
| Interface   | Module      |
| Class       | Class       |
| Struct      | Class       |

TABLE 9.1 .NET to Ruby Object Mapping

| <b>.NET Object</b> | <b>Ruby Object</b>        |
|--------------------|---------------------------|
| Delegate           | Class                     |
| Enum               | Class                     |
| Enum constant      | Class method              |
| Constant           | Class method              |
| Static method      | Class method              |
| Method             | Instance method           |
| Field              | Getter and setter methods |
| Property           | Getter and setter methods |

The “Using .NET Objects” section, later in this chapter, describes how to use each of the .NET objects in IronRuby.

## Types Differences

The CLR and Ruby have different types. For example, `int`, `short`, and `byte` don’t have an equivalent in Ruby. From the other side, `TrueClass`, `Fixnum`, and `Bignum` don’t have an equivalent CLR type.

When .NET objects are converted to IronRuby, most of the CLR types stay the same. For example, `System.String` stays the same in IronRuby (with its naming convention: `System::String`).

However, a few types are mapped to native Ruby types, as shown in Table 9.2.

TABLE 9.2 CLR to Ruby Type Mapping

| <b>CLR Type</b>       | <b>Ruby Type</b>                                 |
|-----------------------|--|
| <code>Int32</code>    | <code>FixNum</code>                              |
| <code>Double</code>   | <code>Float</code>                               |
| <code>Boolean</code>  | <code>TrueClass</code> , <code>FalseClass</code> |
| <code>DateTime</code> | <code>Time</code>                                |

## Coding Standards Collision

Coding standards of .NET code and Ruby code are different. .NET namespaces, classes, and methods are written in Pascal case (with the first letter of each word capitalized). Ruby modules and classes are also written in Pascal case, but the methods are written in lower-case letters.

## PASCAL CASING IN .NET CODE

Pascal case isn't always used in .NET code. For example, nothing stops developers from writing the following code:

```
namespace startWithLowercase
{
    class lowercaseToo
    {
        void lowercaseAgain()
        {
        }
    }
}
```

This kind of naming is problematic in Ruby code. As you may recall, when we discussed classes and modules in Ruby, I mentioned that they must start with a capital letter. Therefore, `class lowercaseToo` is not valid in Ruby.

This kind of convention is rarely used within .NET classes. This is why IronRuby does not support it in version 1.0. Namespaces and classes that start with a lowercase letter are ignored.

---

To solve this naming convention conflict and make .NET code feel more Ruby-like, IronRuby allows calling .NET code with Ruby idioms:

- ▶ CLR namespace, interface, and class names are capitalized. That is, their names actually stay the same.
- ▶ CLR methods can be used with their .NET name (for example, `WriteLine`). They can also be called with a Ruby-like name, via a conversion of Pascal case to lowercase letters with an underscore as a word delimiter (an action known as *mangling*), `write_line`:

```
require "system"
# Parenthesis is optional, just like regular Ruby code
System::Console.WriteLine "Hello!"
# This line works as well:
System::Console.write_line "Hello!"
```

- ▶ CLR methods of IronRuby built-in types that are based on CLR classes such as `Object` or `Fixnum` are not mangled. That is, methods such as `ToString` or `GetType` are not accessible as `to_string` or `get_type`.
- ▶ CLR method names that contain a two-letter word that is not `as`, `by`, `do`, `id`, `it`, `if`, `in`, `is`, `go`, `my`, `of`, `ok`, `on`, `to` or `up` will not be mangled. For example, a method named `ReloadMe` will be available in IronRuby only by `ReloadMe` and not by `reload_me`.

- ▶ CLR method names will not be mangled if they are one of `Class`, `Clone`, `Display`, `Dup`, `Extend`, `Freeze`, `Hash`, `Initialize`, `Inspect`, `InstanceEval`, `InstanceExec`, `InstanceVariableGet`, `InstanceVariableSet`, `InstanceVariables`, `Method`, `Methods`, `ObjectId`, `PrivateMethods`, `ProtectedMethods`, `PublicMethods`, `Send`, `SingletonMethods`, `Taint` or `Untaint`. These are either Ruby keywords or global methods, and mangling their names results in naming collisions.
- ▶ CLR method names that contain only uppercase letters and more than two letters will not be mangled. For example, `SH` will be mangled to `s_h` whereas `SHA` will not be mangled.
- ▶ CLR method named with more than three attached uppercase letters followed by lower case letters will not be mangled. For example, `IPAddress` will be mangled to `ip_address` but `DOTCom` will not be mangled.
- ▶ CLR virtual methods that are overridden in IronRuby must be overridden with their Ruby-like name, which is all lowercase delimited by underscores.

## Private Binding Mode

Generally, all members of a loaded .Net assembly are mapped to IronRuby code. However, in the default behavior, IronRuby doesn't map private members (improves the mapping process performance). Private binding mode can be used to force IronRuby to map private members as well.

To turn on the private binding mode, start IronRuby with the `-X:PrivateBinding` switch.

Instead of `ir %filename%`, the call should be `ir -X:PrivateBinding %filename%`. (Note that `ir` switches are case-sensitive.)

When the private binding switch is applied, IronRuby behavior changes—all protected and private methods are mapped to *public* Ruby methods. Therefore, you can use them freely from inside or outside their classes:

In C#

```
public class Printer
{
    private void PrintHello()
    {
        Console.WriteLine("Hi from C#");
    }
}
```

In IronRuby, run with the `-X:PrivateBinding` switch:

```
Printer.new.PrintHello # Prints "Hi from C#"
# Get Printer class only private methods
Printer.new.private_methods - Class.private_methods
# = []
```



```
# Get Printer class only public methods
Printer.new.public_methods - Class.public_methods
# = ["print_hello"]
```

## Using .NET Objects

We've seen how IronRuby maps .NET types and objects. Now let's look at how they can be used from IronRuby.

### Namespaces

CLR namespaces are converted to modules in IronRuby. When the assembly is required, the module is available through the namespace name.

The namespace-contained objects are accessible using the module name followed by two colons.

For example, the namespace in C#

```
// Namespace
namespace CSharpNamespace
{
    ...
}
```

And the module in IronRuby

```
require "c:/assemblies/customAssembly.dll"
CSharpNamespace.class # = Module
# Accessing a CSharpNamespace object named "SomeObject"
CSharpNamespace::SomeObject
```

### EMPTY NAMESPACES

If a namespace doesn't contain members, the namespace will not exist in IronRuby. Attempts to access it result in an error.

The same thing also happens if a namespace contains private members only.

### Aliasing Namespaces

Namespaces are good for grouping different code objects together, but it can become annoying to write their names before any object they contain.

The first way of dealing with it is by setting an alias for the namespace, giving it a shorter name:

```
require "System"
# Accessing the String class:
System::String
# Giving alias to System
Sys = System
# Now String can be accessed with Sys:
Sys::String
```

It is also possible to remove the need for the namespace at all. Just like using in C# and Import in Visual Basic.Net, the `include` method in Ruby brings the included module to the current scope, which means you can skip writing the module name every time:

```
require "System.Data"
# Accessing the DataSet class:
System::Data::DataSet.new
# Include the System.Data namespace
include System::Data
# Now Data can be accessed without a prefix
DataSet.new
```

Aliasing namespaces or using the `include` method can be done inside and outside code containers (modules, classes, methods, and so on). The effect takes place only within the scope where the request is created, as follows:

```
require "System.Data"
class MyClass
  # Inside the class scope
  include System::Data
  def my_method
    DataSet.new # OK
  end
end
# Outside the scope
DataSet.new # Error!
System::Data::DataSet.new # OK
```

## UNDERSTANDING THE ALIASING SYNTAX

For some, the syntax of aliasing CLR namespace names might be a bit weird. It is similar to setting a variable. Well, it is.

As mentioned, CLR namespaces are converted to modules in IronRuby. Modules in Ruby are objects, like everything else. Therefore, there is no problem setting them to variables.

This is different from C# or VB.Net, for example. In those languages, you should use the `using` or `Import` directive to alias a namespace.

---

## Interfaces

CLR interfaces do not have an equivalent type in the Ruby world. To solution this problem, interfaces are converted to empty modules in IronRuby.

When a class that implements the interface exists, after it is converted to IronRuby, the `ancestors` method returns the interface module as part of the `ancestors` list.

For example, consider the following C# code:

```
public interface CSharpInterface { }
public class CSharpClass : CSharpInterface { }
```

Now let's look into the `ancestors` list in IronRuby:

```
CSharpClass.ancestors # = [CSharpClass, CSharpInterface, Object, Kernel]
```

If `CSharpClass` actually inherits from `CSharpInterface`, it generates major problems when a class implements multiple interfaces, or inherits from a class. Ruby cannot support it because it does not support multiple inheritance. A similar technique that Ruby does support is mixins, which are used in this situation.

CLR interfaces are translated to empty Ruby modules. This is because Ruby doesn't have support for interfaces; its duck typing capabilities make interfaces rather unnecessary. To keep the chain of inheritance, this module is mixed in to the class. When this is done, the application can still check whether a class responds to a given interface, and the problem of multiple inheritance is solved.

## Classes

CLR classes are converted to Ruby classes, which generally means that they stay and act in a similar way.

There is one difference, though. When the IronRuby interpreter is run with the `-X:PrivateBinding` switch, classes that are marked as `internal` become accessible just as if they were public classes.

In C#

```
public class A { }
internal class B { }
```

In IronRuby

```
A.new # OK
B.new # OK only if run with -X:PrivateBinding. Otherwise, error.
```

## Structs

CLR structs are very similar to classes in .NET languages, so it is natural that they will be converted to classes in IronRuby.

The difference between a struct and a class is that a struct is a value-type, whereas a class is a reference-type. This means that, for example, when struct is passed to a method, no matter what modifications this method makes to it, those modifications will not affect the original struct.

As a result of the mapping of structs to classes, this behavior doesn't happen in IronRuby; CLR structs behave like every other class:

In C#

```
public struct FullName
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

In IronRuby

```
struct = FullName.new
struct.FirstName = "Shay"
struct.LastName = "Friedman"
puts "#{struct.FirstName} #{struct.LastName}"
# Prints "Shay Friedman"

# The next method is a method which changes the
# struct internally. It will not affect the original
# struct in C# or VB.Net but will work regularly in IronRuby
def change_struct(struct)
    struct.FirstName = "John"
    struct.LastName = "Doe"
end
change_struct(struct)
puts "#{struct.FirstName} #{struct.LastName}"
# Prints "John Doe"
```

## Delegates

Delegates on the CLR are method signatures that are used as pointers to methods with the same signature.

For example, in the following C# code, I define a delegate and a method that receives the delegate and executes it:

```

class Printer
{
    public delegate void PrintValue(string value);
    public void Print(PrintValue printValue)
    {
        printValue("Hello");
    }
    // Sample use of the delegate
    public void TestPrint()
    {
        // Use an anonymous method as a delegate
        Print(delegate(string str)
            { Console.WriteLine(str); }); }
}

```

A common Ruby technique might pop into your head right away: blocks. Delegates are, indeed, very similar in concept to blocks. Anonymous methods, which are used in the preceding sample, are even closer to Ruby's blocks.

However, there are a few major differences between them:

- ▶ Delegates have a name. Blocks, procs, and lambdas in Ruby are anonymous.
- ▶ Blocks (not procs nor lambdas) can appear only as the last parameter of methods. Delegates are free of this restriction.
- ▶ Delegates are invoked using the `invoke` method. Block, proc, and lambdas do not support that. They are invoked using `yield`, `call`, or square brackets.

As a result of these differences, none of the different Ruby blocks can replace the CLR delegates. Consequently, delegates are converted to classes that receive blocks on initialization. After the class is initialized, the class `invoke` method executes the block.

Assuming we have required the C# code from the previous sample, the next code creates a `PrintValue` delegate implementation and uses it:

```

dlg = Printer::PrintValue.new { |str| puts str }
# We can pass it to the Print method:
Printer.new.Print(dlg) # Prints "Hello"
# The delegate can also be invoked directly:
dlg.invoke("Hi there!") # Prints "Hi there!"

```

## Events

Events are a common practice in the .NET Framework. They provide a way to sequentially invoke .NET delegates that match a specific signature.

The two basic concepts of events are subscribing and unsubscribing them. When a code block is subscribed to an event, it is executed every time the event is raised. When there is no need to react to the event anymore, the code block can be unsubscribed from the event.

IronRuby provides several ways to use .NET events, even though the Ruby language doesn't have built-in support for it.

Assume the next code exists on the events.dll file. It is C# code and has an event, `MyEvent`, and a method that raises it. The event uses the built-in delegate, which consists of two parameters, the event sender object and the event arguments (a `MyEventArgs` object):

```
public class EventsSample
{
    public class MyEventArgs : EventArgs
    {
        public string Message { get; set; }
        public MyEventArgs(string message)
        {
            Message = message;
        }
    }

    // The event will be sent with the MyEventArgs class as its
    args parameter
    public event EventHandler<MyEventArgs> MyEvent;

    public void InvokeEventSubscribers()
    {
        if (MyEvent != null)
        {
            MyEvent(this, new MyEventArgs("Hello from .Net"));
        }
    }
}
```

### Subscribing Events

IronRuby features a special way to subscribe to .NET events: It provides a new method for that matter—the `add` method.

When you subscribe to an event, a code block that handles the event should be passed. IronRuby offers several ways to define a code block: methods, procs, lambdas, and associated code blocks. All of these can be used to handle CLR events.

The only requirement is that they match the event delegate signature:

```
require "events.dll"

def event_handler(sender, args)
  puts "Method: #{args.message}"
end

handler_method = method(:event_handler)
handler_proc = proc { |sender, args| puts "Proc: #{args.message}" }
handler_lambda = lambda { |sender, args| puts "Lambda: #{args.message}" }

obj = EventsSample.new
# Subscribe with the method
obj.my_event.add handler_method
# Subscribe with the proc
obj.my_event.add handler_proc
# Subscribe with the lambda expression
obj.my_event.add handler_lambda
# It is possible to subscribe twice with the same code block:
obj.my_event.add handler_lambda

# Invoke the subscribers
obj.invoke_event_subscribers
# Prints: "Method: Hello from .Net
#         Proc: Hello from .Net
#         Lambda: Hello from .Net
#         Lambda: Hello from .Net"
```

The exception here is the associated code block. It is possible to subscribe to an event using a block, but it is done without the preceding `add` method. It is simply associated with the event name:

```
obj.my_event { |sender, args| puts "Block: #{args.message}" }
obj.invoke_event_subscribers # Prints "Block: Hello from .Net"
```

### Unsubscribing Events

You unsubscribe to events in the opposite way you subscribe to them: with the `remove` method.

Note that it is not possible to unsubscribe an associated code block. If you need to unsubscribe, use the other alternatives to subscribe and unsubscribe from the event.

```
obj.my_event.remove handler_method
obj.my_event.remove handler_proc
obj.my_event.remove handler_lambda
```

```

obj.my_event.add handler_lambda
obj.invoke_event_subscribers
# Prints: "Method: Hello from .Net
#         Proc: Hello from .Net
#         Lambda: Hello from .Net
#         Lambda: Hello from .Net"

# Unsubscribe
obj.my_event.remove handler_method
obj.my_event.remove handler_proc
obj.my_event.remove handler_lambda

obj.invoke_event_subscribers
# Prints: "Lambda: Hello from .Net"

```

## Enums

Ruby does not have enums within its arsenal. Therefore, enums are translated to classes. These classes include class methods, which are the enum values.

As a result, accessing enum values is identical to .NET syntax:

In C#

```

public enum CSharpEnum
{
    EnumConstant1,
    EnumConstant2
}

```

In IronRuby

```

# Accessing enum values:
CSharpEnum.EnumConstant1

```

IronRuby also supports combining several enum values, which are marked as flags. This is done with the logical OR operator (`|`):

In C#

```

[Flags]
public enum CSharpFlaggedEnum
{
    EnumConstant1,
    EnumConstant2
}

```



In IronRuby

```
# Combining multiple flag values:
enum_value = CSharpFlaggedEnum.EnumConstant1 | CSharpFlaggedEnum.EnumConstant2
```

## Constants

CLR constants are translated to getter methods. Public and protected constants are mapped, but private constants are not mapped to IronRuby at all.

In C#:

```
public class ConstsClass
{
    public const string PublicConst = "I'm public";
    protected const string ProtectedConst = "I'm protected";
    private const string PrivateConst = "I'm private";
}
```

In IronRuby:

```
ConstsClass.PublicConst # = "I'm public"
ConstsClass.ProtectedConst # = "I'm protected"
ConstsClass.PrivateConst # Error!
```

## UNDERSTANDING CONSTANTS MAPPING

To some, constants and enum constants mapping to class methods might seem irrational because Ruby has constants, too. The answer to that is simple. In Chapter 5, “Ruby Basics,” we discuss the special behavior of Ruby constants: They are changeable. This is entirely different from the behavior of .NET enum constants, which cannot be changed.

To conform with .NET’s behavior, constants are converted to getter methods without setter methods.

## Methods

Methods are the core building block of both .NET languages and Ruby. As a result, translating methods from one language to the other is simple. It is important, however, to be familiar with the different cases and to know what to expect.

### Instance Methods

Instance methods are translated to Ruby instance methods.

Public methods are the most direct case; they are mapped to public Ruby methods.

Protected methods are mapped to protected Ruby methods. In this case, you should pay attention to the difference between protected methods in C# or VB.Net and protected methods in Ruby. In Ruby, protected methods are accessible by all instances of the class, not by the specific deriving class only.

Private methods are not mapped at all. You cannot see them use them unless private binding mode is turned on.

### Static Methods

CLR static methods are converted to Ruby class methods. Same rules apply here for public, protected, and private static methods. The private binding mode has the same effect on static methods as it has on regular methods.

### Overloaded Methods

Ruby doesn't support method overloading. Ruby's approach to overloading is optional parameters: If you have optional parameters, you can define default values for them.

IronRuby deals with that by inspecting the input to the method and sending the request to the right CLR method. From the developer perspective, the method is just like any other method with optional arguments:

In C#

```
public class OverloadingSample
{
    public OverloadingSample()
    {
        Console.WriteLine("No arguments");
    }
    public OverloadingSample(string a) : this()
    {
        Console.WriteLine("1 argument");
    }

    public void Method()
    {
        Console.WriteLine("No parameters");
    }
    public void Method(string a)
    {
        Console.WriteLine("1 parameter");
    }
}
```

In IronRuby

```
c1 = OverloadingSample.new # Prints "No arguments"
c2 = OverloadingSample.new("hi") # Prints "1 argument No arguments"
```

```
c1.Method # Prints "No parameters"
c1.Method("hi") # Prints "1 parameter"
```

The mapping to the right CLR method is not based solely on the number of arguments, but on the parameter types, too. It is done this way to handle overloaded methods with the same number of arguments and different argument types:

In C#

```
public class OverloadingSample
{
    public void Method(string a, string b)
    {
        Console.WriteLine("2 strings");
    }
    public void Method(string a, int b)
    {
        Console.WriteLine("string + int");
    }
}
```

In IronRuby

```
c = OverloadingSample.new
c.Method("hi","hi") # Prints "2 strings"
c.Method("hi", 3) # Prints "string + int"
c.Method("hi", Time.new) # Error!
```

### Indexer Methods

Indexers in .NET have a unique syntax. Indexers in Ruby are like every other method; they just have a more special name than the others.

The mapping here is simple. The CLR indexer getter part is mapped to the `[]` method in Ruby, and the indexer setter part is mapped to the `[]=` method:

In C#

```
public class IndexerSample
{
    public int this[int x]
    {
        get { return x + 1; }
        set { /* ... code ... */ }
    }
}
```

In IronRuby

```
c = IndexerSample.new
c[1]
c[17] = 8
c.methods.delete_if { |x| x =~ /[]?/ } # = [ [], []= ]
```

### Special .NET Methods

Every IronRuby object contains, like every .NET object, two methods: `GetHashCode` and `Equals`. These methods are used for several CLR operations, such as comparison, hash-related algorithms (like in `Dictionary` or `HashTable` objects), and more. The Ruby language has similar methods that are used for the exact same aim: `hash` and `eq?`.

This similarity generates a conflict. If a developer wants to write an IronRuby object that implements a special hash calculation, what should be implemented? `GetHashCode` or `hash`? IronRuby takes this into consideration and does an automatic mapping between the CLR and the Ruby methods.

This means that when in the Ruby world, we need to stick to Ruby methods and implement `hash` or `eq?` and not `GetHashCode` or `Equals`.

Inside the IronRuby context, only the `hash` and `eq?` methods should be implemented and called. Note that by implementing these methods, calling the CLR methods will not use the new implementations, as follows:

```
class String
  def hash
    1111
  end
end

str = "Ruby string!"
puts str.hash # = 1111
puts str.GetHashCode # = -1632647123
```

Outside the IronRuby context (when accessing IronRuby objects from C#, for example), the `GetHashCode` and `Equals` methods are seamlessly mapped to the `hash` and `eq?` methods accordingly.

Note that the `==` operator is not related to these mappings because it doesn't have the same behavior. The `==` operator compares the object contents, and `eq?` compares the object hashes.

### Special Argument Types

.NET features three special argument types: `out`, `ref`, and `params`. These appear only partially in Ruby, so a special treatment is required.

**out parameter** This parameter type is used to return multiple values to the method caller. The method is also required to assign a value to the parameter before terminating. Ruby does not support out parameters, so the mapping is done with a trick: out parameters are mapped in IronRuby to return values. A CLR method with out parameters returns an array of values, including the original return type and the out parameters. Trying to pass objects to out parameters can result in an error because the method will not expect them:

In C#

```
public class OutSample
{
    public string OutMethod(string value, out string outParam)
    {
        outParam = "You're out!";
        return "Done";
    }
    public string OutMethod2(string val, out string outParam, string val2)
    {
        outParam = "You're out!";
        return "Done";
    }
}
```

In IronRuby

```
c = OutSample.new
c.OutMethod("hi") # = [ "Done", "You're out!" ]

# Can be used in a convenient way:
ret_val, out_param = c.OutMethod("hi")

# Cannot be used like in .Net languages:
str = ""
c.OutMethod("hi", str) # Error!

# Pay attention that out parameters are excluded from the
# method signature in IronRuby
c.OutMethod2("value1", "value2") # = [ "Done", "You're out!" ]
```

**ref parameter** This parameter type allows sending value-type variables to methods, changing them inside, and using the new value outside the method (that is, making value-types act like reference-types). Unlike out parameters, ref parameters must be initialized before they are passed to the method. In IronRuby, ref arguments, similar to out arguments, are returned from the method as an array. The difference from out arguments is that ref parameters should be passed to the method, too, because they might be needed inside the method. (Remember, they are already initialized when they are passed to the method.)

Note that this solution makes `ref` parameters in IronRuby lose their initial aim; they are not modified by the method. If you need to change them, make sure to set them with the matching return value.

When `ref` and `out` parameters are used on the same method, the returned array contains their values in the order they appear on the method signature:

In C#

```
public class OutRefSample
{
    public string OutRefMethod(string val1, out string outParam,
                              ref string refParam, string val2,
                              out string outParam2)
    {
        outParam = "out 1";
        refParam = "ref 1";
        outParam2 = "out 2";
        return "Done";
    }
}
```

In IronRuby

```
c = OutRefSample.new
ret_val, out1, ref1, out2 = c.OutRefMethod("val1", "ref val", "val2")
```

**params parameter** This parameter type allows sending an arbitrary number of arguments to the method. The `params` parameter must be the last one in the parameters list.

`params` parameters are close to `*` parameters in Ruby. This is exactly the mapping that takes place:

In C#

```
public class ParamsSample
{
    public void ParamsMethod(params string[] args)
    {
        Console.WriteLine(args.Length);
    }
}
```

In IronRuby

```
c = ParamsSample.new
c.ParamsMethod("val1", "val2", "val3") # Prints "3"
```

## Fields

CLR fields are different from instance or class variables in IronRuby. CLR fields can be marked as public and then be accessed from outside the class. In Ruby, class and instance variables are accessed only from their class and cannot be accessed from outside. Getter and setter methods should be written to make them accessible to the outside scope.

It is important to notice that IronRuby mangles the names of fields, too. However, it mangles only field names that are Pascal cased. Fields are not written in Pascal case according to the .NET coding conventions. As a result, it is recommended to use fields in IronRuby in their original case to avoid mistakes.

To bridge this gap, CLR fields are translated to getter and setter methods in IronRuby. Public and protected fields are accessible like every other public method. Private fields are accessible only in private binding mode.

In C#

```
public class FieldsSample
{
    public string publicField;
    protected string protectedField;
    private string privateField;
}
```

In IronRuby

```
c = FieldsSample.new
c.publicField = "Hi"
c.publicField # = "Hi"
c.protectedField = "Hi"
c.protectedField # = "Hi"
c.privateField # Error!
c.methods.select { |x| x =~ /[a-z]*Field?/ }
# = [ publicField=, publicField, protectedField=, protectedField ]
```

## Properties

CLR properties are mapped to getter and setter methods in IronRuby. Properties' access control levels are mapped, too:

- ▶ Public properties are mapped to Ruby public getter and setter methods.
- ▶ Protected properties are translated to Ruby protected getter and setter methods. Notice that Ruby's protected access control is not CLR's protected access control.
- ▶ Private properties are not mapped at all when not in private binding mode.

- ▶ Mixed access control levels for the getter and setter parts on the same property are mapped accordingly to the getter and setter methods in IronRuby.
- ▶ Properties with only getter or setter parts are mapped only with the matching method.

In C# (C# 3.0 syntax)

```
public class PropertiesSample
{
    public string PublicProperty { get; set; }
    protected string ProtectedProperty { get; set; }
    private string PrivateProperty { get; set; }

    public string Mixed1 { get; protected set; }
    protected string Mixed2 { private get; set; }
}
```

In IronRuby

```
c = PropertiesSample.new
c.PublicProperty; c.PublicProperty = "hi"
c.ProtectedProperty # Error! Can be done only from within the class
c.instance_eval "ProtectedProperty = 'hi'" # OK!
c.PrivateProperty # Error!
c.Mixed1
c.Mixed1 = "hi" # Error! Only from within the class
c.Mixed2 # Error!
c.Mixed2 = "hi" # Error! Only from within the class
```

### Private Binding Mode

When IronRuby runs in private binding mode, all properties, regardless of their visibility level, are treated as public.

For example, all the errors that occurred in the preceding sample code wouldn't occur in private binding mode.

## Generics

Generics is a commonly used concept that is a vital part of the .NET Framework. Ruby, however, is dynamically typed. As a result, a generics-like capability is not needed, and therefore it is not a part of the language.

IronRuby supports .NET generic objects and allows using them in a slightly different syntax.

The way to use generic objects is by passing the types between square brackets, [ ]. For example, using the `System.Collections.Generic.List` class to create a list of strings:



```
include System::Collections::Generic
list = List[String].new
list.add("IronRuby!")
list.add(4) # Error: "can't convert Fixnum into String"
```

If multiple types should be passed, the types are separated with a comma inside the square brackets (for example, the Dictionary class):

```
include System::Collections::Generic
# There is no problem with using Ruby and CLR types:
Dictionary[Fixnum, System::Int64].new
```

Calling generic methods also has a special syntax. It consists of getting the method object, supplying it with the generic types, and calling it.

To get the method object, we need Ruby's reflection method method. When the method object exists, we need to supply the generic types with the of method. This results in a method object of the types we supplied to it. All there is left is to execute it like any other method object—via the call method.

For example, assume we have the next C# methods:

```
public class GenericsDemoClass
{
    public string Test<T>(T param1)
    {
        return param1.ToString();
    }
    public string Test<T,S>(T param1, S param2)
    {
        return param1.ToString() + "," + param2.ToString();
    }
}
```

The next code snippet executes the preceding generic methods from IronRuby:

```
generic_obj = GenericsDemoClass.new
# Get the method object
test_method = generic_obj.method(:test)
# Get the type-specific method object
string_text_method = test_method.of(String)
# Execute the generic method
string_text_method.call("IronRuby") # Returns "IronRuby"
# One line call:
generic_obj.method(:test).of(String).call("IronRuby")
```

```
# To call a method with multiple generic types, just pass as  
# much types to the of method as needed  
generic_obj.method(:test).of(String, Fixnum).call("IronRuby", 1)  
# Returns "IronRuby,1"
```

## Special IronRuby Methods

The preceding section covered the mapping IronRuby does to CLR objects to make them conform to their new environment. This mapping is essential, but it has one disadvantage: It makes CLR objects look different than their real implementation.

To deal with that, IronRuby adds unique methods that are intended to help communicate better with the CLR. These methods allow interacting directly with CLR types without the limits of the IronRuby mapping.

### Object Class Methods

IronRuby adds methods that are available applicationwide by adding them to the `Object` class.

#### **to\_clr\_type**

IronRuby core objects are mapped to CLR objects. For example, the `Thread` class is mapped to the `System.Threading.Thread` class. On regular Ruby methods, like `class`, the IronRuby `Thread` class acts like the original Ruby `Thread` class:

```
Thread.class # = Class
```

By using `to_clr_type`, the `Thread` class CLR mapping will be discovered:

```
Thread.to_clr_type # = System.Threading.Thread
```

In case the method is used with a pure Ruby object, `nil` will be returned:

```
class MyRubyClass; end;  
MyRubyClass.to_clr_type # = nil
```

#### **clr\_member**

The `clr_member` method receives a symbol of the method name and returns the CLR method:

```
trim = System::String.new(" Hello ").clr_member(:Trim)  
# Invoke the method  
puts trim.call # Prints "Hello"
```

There are two main uses for this method. The first is to inspect the original CLR method (by using the special `Method` class methods discussed later in this section). The second is to make sure the original CLR method is used and not its redefinition.

For example, the following sample is possible in IronRuby:

```
class System::String
  def Trim
    # Sabotage Trim functionality
    "<<" + self + ">>"
  end
end

str = System::String.new(" Hello ")
puts str.Trim # Prints "<< Hello >>"
puts str.clr_member(:Trim).call # Prints "Hello"
```

The `clr_member` method is available both as a class method and as an instance method. When it is used as a class method, static CLR methods are searched for. When it is used as an instance method, instance CLR methods are looked for.

## Class Class Methods

Classes differentiate from other objects by their capability to be initialized. IronRuby adds special constructor-related methods to its classes for investigating and using CLR constructors rather than IronRuby ones.

### **clr\_constructor and clr\_ctor**

IronRuby tends to modify CLR constructors and fit them to its conventions. For example, classes with overloaded constructors are mapped to a single `new` method in IronRuby.

There is another issue with CLR constructors: A few classes in IronRuby are mapped directly to CLR classes. This makes it impossible to call the real CLR constructor (for example, the `Thread` class):

```
System::Threading::Thread.new
# Error: "must be called with a block"
```

This is not the expected result. The CLR `Thread` class expects a `ThreadStart` delegate, not a Ruby block. This happens because IronRuby's `Thread` class is, by design, a direct mapping of the CLR `Thread` class:

```
System::Threading::Thread == Thread # = true
```

To deal with these issues, `clr_constructor` and its alias, `clr_ctor`, can be used. These methods return a `Method` object representing the constructor. The `Method` object can then

be further investigated or called. Later in this chapter I discuss the special methods that are added to the Method class.

In C#

```
public class OverloadingSample
{
    public OverloadingSample()
    {
    }
    public OverloadingSample(string a) : this()
    {
    }
}
```

In IronRuby

```
OverloadingSample.clr_ctor.class # = Method
OverloadingSample.clr_ctor.clr_members.size # = 2
```

It is possible to also execute the constructor with the call method:

```
OverloadingSample.clr_ctor.call # Executes the parameterless constructor
OverloadingSample.clr_ctor.call("hello") # Executes the second constructor
```

### **clr\_new**

Similar to the `clr_constructor` method, `clr_new` deals with CLR constructors. However, they have different behaviors. `clr_new`, unlike `clr_constructor`, executes the CLR constructor right away. Notice the different behavior of the Thread class when initiated with `clr_new`:

```
System::Threading::Thread.clr_new
# Error "Wrong number of arguments"
```

With the `clr_new` method, the constructor is looking for a CLR `ThreadStart` method and not a Ruby block.

## **Method Class Methods**

The Method class has two special CLR-related methods. These methods are targeted to examine a CLR method, especially its overloads list.

### **clr\_members**

The `clr_members` method returns an array of `System.Reflection.RuntimeMethodInfo` objects. Each array item represents a single overload of the method:

```

str = System::String.new("Hello")
trim = str.clr_member(:Trim)
trim.clr_members.length # = 2
trim.clr_members[0].ReturnType # = System.String
trim.clr_members[0].GetParameters().Length # = 1
trim.clr_members[1].GetParameters().Length # = 0

```

A nice feature of the `RuntimeMethodInfo` class is its string representation of the method signature. Printing out the `clr_members` result array can provide a good overview of the possible method overloads and their required parameters:

```

str = System::String.new("Hello")
puts str.clr_member(:Trim).clr_members
# Output:
# System.String Trim(Char[])
# System.String Trim()

```

### overload

Sometimes there is a need for only a specific overload and not for the whole list. This is exactly what the `overload` method is for. It receives types as parameters, which represent the argument list of the requested overload, and returns the matching `Method` object.

For example, the following code sample retrieves the `String.Replace(String, String)` method overload and then the `String.Replace(Char, Char)` overload:

```

str = System::String.new("Hello")
str.clr_member(:Replace).overload(System::String, System::String)
str.clr_member(:Replace).overload(System::Char, System::Char)

```

## String Class Methods

IronRuby strings are not mapped directly to CLR `String` objects as you might expect. IronRuby strings are mapped to an object called `MutableString`, which is a part of the IronRuby implementation.

As a result of this condition, IronRuby provides a method to easily convert an IronRuby string to CLR string, `to_clr_string`:

```

str = "IronRuby!"
str.class # = String
str.class.to_clr_type # = IronRuby.Builtins.MutableString
str.to_clr_string.class # = System.String

```

## TO\_CLR\_STRING IS A NONDESTRUCTIVE OPERATION

`to_clr_string` is a nondestructive operation and produces a new CLR string out of the current `MutableString`.

## The IronRuby Class

IronRuby comes with a unique class targeted to provide IronRuby-specific environment information and operations. As such, you find it useful in cases when you need to dynamically investigate the environment or IronRuby's behavior.

The class provides six methods and an inner class, as shown in table 9.3.

TABLE 9.3 IronRuby Class Members

| Member Name       | Description   |
|-------------------|---|
| configuration     | <p>A <code>DlrConfiguration</code> object with the current language configuration.</p> <p>Sample:</p> <pre># Gets a value indicating whether private # binding mode is turned on IronRuby.configuration.private_binding # Get a value indicating whether debug # mode is turned on IronRuby.configuration.debug_mode</pre>  |
| globals           | <p>Returns a <code>Microsoft.Scripting.ScopeStorage</code> object that represents the global scope. It provides methods to get, set, and remove global objects.</p> <p>Sample:</p> <pre>class Demo; end; # Get all global members IronRuby.globals.get_member_names # Returns ['Demo'] # Create a scope variable IronRuby.globals.set_value("demo", true, Demo.new) # Get the variable value IronRuby.globals.get_value("demo", true) # Remove the variable IronRuby.globals.delete_value("demo", true)</pre> |
| loaded_assemblies | <p>Returns an array of <code>Assembly</code> objects representing the loaded CLR assemblies.</p> <p>Sample:</p> <pre>IronRuby.loaded_assemblies.each do  assembly    puts assembly.full_name end # prints: # mscorlib, Version=2.0.0.0... # System, Version=2.0.0.0...</pre>  |

TABLE 9.3 IronRuby Class Members

| Member Name    | Description  |
|----------------|--|
| loaded_scripts | <p>Returns a read-only dictionary with name-scope pairs representing scripts that were loaded using <code>IronRuby.require</code> or <code>IronRuby.load</code> methods.</p> <p>Sample:</p> <pre>IronRuby.load('C:\demo.rb') IronRuby.loaded_scripts.each do  name, scope    puts name end # prints "C:\demo.rb"</pre>   |
| require        | <p>Same as <code>Kernel.require</code> that is discussed previously in this chapter but returns the assembly object or script scope of the loaded object.</p> <p>Sample:</p> <pre># Returns nil: require 'C:\MyAssembly.dll' # Returns Assembly object assembly = IronRuby.require 'C:\MyAssembly.dll' puts assembly.full_name # Prints "MyAssembly, Version=1.0.0.0..."</pre>   |
| load           | <p>Same as <code>Kernel.load</code> that is discussed previously in this chapter but returns the assembly object or script scope of the loaded object.</p>   |
| IronRuby::Clr  | <p>A class that contains IronRuby-specific CLR modules like <code>BigInteger</code>, <code>Float</code>, or <code>String</code>. These have no specific purpose, but they might give you an idea of the CLR types that IronRuby types are based on.</p> <p>An additional and interesting class is the <code>IronRuby::Clr::Name</code> class. This class contains name-mangling methods. For example, if you want to check how a method name is mangled (or unmangled—the reverse operation), you can use this class:</p> <pre>IronRuby::Clr::Name.mangle("HelloWorld") # returns "hello_world" IronRuby::Clr::Name.mangle("AAA") # returns nil (meaning that this method # name is not mangled) IronRuby::Clr::Name.unmangle("hello_world") # returns "HelloWorld"</pre> <p>The <code>mangle</code> method can also be accessed via <code>clr_to_ruby</code> and the <code>unmangle</code> method via <code>ruby_to_clr</code>.</p> <p>It is also possible to create an instance of the <code>Name</code> class and then investigate the name mangling results:</p> <pre>name = IronRuby::Clr::Name.new("HelloWorld") name.ruby_name # = "hello_world" name.clr_name # = "HelloWorld"</pre> |

## CLR Objects and Ruby's Reflection

IronRuby allows investigating objects—getting a list of the available methods, variables, execute dynamic code, and more. Chapter 8, “Advanced Ruby,” covers reflection in more detail.

In IronRuby, CLR objects are also available for Ruby's reflection operations:

```
# Get System.String static methods:
System::String.methods - Class.methods
# Output: ['join',..., 'is_null_or_empty', 'compare',...,
# 'format', ..., 'empty']

# Get System.String instance methods:
System::String.instance_methods - Class.instance_methods
# Output: ['equals',..., 'to_char_array',..., 'starts_with'...]
```

## The Basic Object

In Ruby, everything is an object. `Object` is the base class that everything inherits from. Changing the `Object` class affects every single object across the application. This concept is identical to C# and Visual Basic .NET, where `Object` is also the root of all objects.

One of the most important keys to understanding the interoperability of IronRuby and the .NET Framework is their shared basic object: *IronRuby's Object is actually System.Object!*

To demonstrate that, look at the following code line:

```
Object == System::Object # = true
```

This doesn't happen because of a similar name. The types are actually equal.

Passing around objects between .NET and IronRuby makes more sense now. It works because in .NET, IronRuby objects are eventually `System.Object`, and in IronRuby, .NET objects are eventually `Object`.

## Summary

IronRuby expertise is the seamless integration with .NET objects. In this chapter, you learned the fundamentals of this expertise. You learned about loading .NET assemblies, using the different types of objects and the new features that IronRuby adds to .NET objects and to the Ruby environment.

The next chapter takes you one step further in IronRuby's .NET interoperability. You learn how to integrate and implement .NET objects in your IronRuby code.



*This page intentionally left blank*

## CHAPTER 10

# Object-Oriented .NET in IronRuby

**.NET** languages are designed from the beginning to be fully object-oriented, as is Ruby.

They support inheritance, method overriding, access control, and more. As a part of its interoperability with the CLR, IronRuby allows Ruby code to act similarly to .NET code and implement object-oriented concepts with .NET objects, the Ruby way.

In this chapter, you learn how to inherit from .NET classes and interfaces, use Ruby object-oriented concepts with CLR types, and become familiar with the rough edges, like sealed CLR classes.

## Inheriting from CLR Classes

Inheritance is a vital part of .NET languages, and IronRuby, too. As you saw in Chapter 6, “Ruby’s Code-Containing Structures,” Ruby has inheritance capabilities with Ruby classes. This is also possible with CLR classes.

In the following subsections, I cover different types of CLR classes and explain how they can be inherited in IronRuby.

### Regular Classes

Regular CLR classes are the simplest case. They act as the superclass of their inheriting classes, with no special catches. Protected members are available to the inheriting classes, as well:

## IN THIS CHAPTER

- ▶ Inheriting from CLR Classes
- ▶ Inheriting from CLR Structs
- ▶ Inheriting from CLR Interfaces
- ▶ Overriding Methods
- ▶ Overriding Properties
- ▶ Overriding Events
- ▶ Opening CLR Classes

In C# (human.dll)

```
public class Human
{
    public void SayHello()
    {
        Console.WriteLine("Hello");
    }
    protected string GetInternalCondition()
    {
        return "all good";
    }
}
```

In IronRuby

```
require "Human.dll"
# Inherit from the CLR Human class
class Doctor < Human
    attr_accessor :first_name, :last_name

    def introduce
        say_hello
        puts "I'm Dr. #{last_name}"
    end
    def check_condition
        if get_internal_condition != "all good"
            puts "Call an ambulance!"
        else
            puts "All is good!"
        end
    end
end

doctor = Doctor.new
doctor.first_name = "John"
doctor.last_name = "Doe"

doctor.introduce # Prints "Hello
                #       I'm Dr. Doe"
doctor.check_condition # Prints "All is good!"
```

Private members are available only in private binding mode. For more information about private method mapping, see Chapter 9, “.NET Interoperability Fundamentals.”

### Constructor Inheritance

In .NET languages like C#, when you inherit from a class without a default constructor (a constructor with no parameters), you have to implement a constructor on the inheriting class and call the constructor of the parent class.

Ruby has a different approach to constructor inheritance. When a superclass has an `initialize` method, it is automatically inherited to the class itself. It doesn't matter whether the method requests none or multiple arguments.

When a class inherits from a CLR superclass, the class automatically inherits the constructor, and there's no need to explicitly define it.

In C# (human.dll)

```
public class Human
{
    public string FirstName { get; set; }

    public Human(string firstName)
    {
        FirstName = firstName;
    }
}
```

In IronRuby

```
require "Human.dll"
class Doctor < Human; end

doctor = Doctor.new # Error "wrong number of arguments"
doctor = Doctor.new("Shay") # OK!
```

### Generic Classes

As a result of the lack of support for generics in Ruby, generic CLR classes are an exceptional case. Because of this limitation, new generic classes cannot be created with IronRuby.

IronRuby takes a special approach to handle the situation, which provides a partial solution to the problem. You can inherit from a specific class type, like `List<String>`, and not from the generic type, `List<T>`:

In C# (human.dll)

```
public class HumanList<T> where T : Human
{
    public void Add(T human)
    {
        // ...Add human to list...
    }
}
```

In IronRuby

```
require "Human.dll"
class DoctorList < HumanList[Doctor]
end

doctors = DoctorList.new
doctors.add(Human.new) # Error! Wrong type!

doctors.add(Doctor.new("John"))
```

## Abstract Classes

Abstract classes are classes that contain abstract code parts that are implemented by the deriving classes. Abstract classes cannot be initialized either.

In IronRuby, abstract classes do not exist because there is no concept of overriding methods. When CLR abstract classes are mapped to IronRuby, they become regular classes that cannot be initialized. Consequently, inheriting from them is just like every other class.

In C# (human.dll)

```
public abstract class Creature
{
    public void Breathe()
    {
        Console.WriteLine("Breathing...");
    }
}
```

In IronRuby

```
require "Human.dll"
class Dog < Creature
    def introduce
        puts "Woof woof!"
    end
end

dog = Dog.new
dog.breathe # Prints "Breathing..."
dog.introduce # Prints "Woof woof!"
dog.class.superclass # = Creature
Creature.new # Error!
```

## Sealed and Static Classes

Sealed and static CLR classes are special: They cannot be inherited. The Ruby language doesn't support such a behavior. IronRuby, however, goes the extra mile and disallows inheriting from sealed and static CLR classes.

Pay attention to the timing of the error. The error will not be raised on the class definition, only when the class is initialized.

In C# (human.dll)

```
public sealed class Man : Human
{
}
```

In IronRuby

```
require "Human.dll"
class MultitaskMan < Man
end
# So far so good, now let's create an instance:
new_man = MultitaskMan.new("Shay") # Error!
```

## Inheriting from CLR Structs

In .NET languages, inheriting for structs is impossible. In IronRuby, we know that structs are actually classes, so you may think that this might become possible.

Structs act like sealed CLR classes in IronRuby. Therefore, even though it might sound logical, they cannot be inherited.

## Inheriting from CLR Interfaces

As you learned in Chapter 9, CLR interfaces are mapped to modules in IronRuby. This mapping makes interface implementation in IronRuby tricky because modules cannot act as superclasses:

In C# (creature.dll)

```
public interface ILivingCreature
{
    void Eat();
}
```

In IronRuby

```
require "creature.dll"
class Dog < ILivingCreature
end
# Error: "superclass must be a Class (Module given)"
```

You also learned in Chapter 9 that interfaces are used as mixins when CLR mapped classes have implemented them. This is the way IronRuby classes can implement interfaces—by using them as mixins.

The following code sample shows how to implement CLR interfaces in IronRuby. I implement the `ILivingCreature` interface in an IronRuby class and pass it to a CLR method that uses `ILivingCreature` objects.

In C# (`creature.dll`)

```
public interface ILivingCreature
{
    void Eat();
}

public static class Runner
{
    public static void FeedCreature(ILivingCreature creature)
    {
        creature.Eat();
    }
}
```

In IronRuby

```
require "creature.dll"
class Cat
  include ILivingCreature # Include interface mixin
  # Implement the interface method
  # Can be implemented by its Ruby name - eat
  # or its CLR name - Eat
  def eat
    puts "Mew Yummy Yummy"
  end
end

Runner.feed_creature Cat.new # Prints "Mew Yummy Yummy"
```

## Overriding Methods

Overriding methods is the center of the object-oriented programming principle: polymorphism. By changing method implementation in different classes, the classes still follow the general flow and rules, but they are still unique. For example, in our `Creature` class, we can define a `Sleep` method. The `Human` class implements this method differently than the `Horse` class. At the end, we have various creatures that sleep, each in its individual way.

The CLR features different methods that can be inherited. In this section, I introduce each of them and demonstrate how to implement them with IronRuby.

### Virtual Methods

Virtual methods are methods that have an implementation, but the inheriting classes can override this implementation. The implementation in a method that is marked as `virtual` is called *default implementation*.

In C#, you have to use the `override` keyword to override a method. In IronRuby, there is no need for that; you just implement the method, using the same name. If the method is not implemented on the inheriting class, the default implementation is used.

In C# (creature.dll)

```
public class Creature
{
    public virtual void Fly()
    {
        Console.WriteLine("Sorry, I can't fly");
    }
}
```

In IronRuby

```
require "creature.dll"
class Bird < Creature
    # Override default implementation
    def fly
        puts "Flying"
    end
end
class Dog < Creature
end
```

```
Bird.new.fly # Prints "Flying"
```



```
Dog.new.fly # Uses default implementation -
           # Prints "Sorry, I can't fly"
```

You can also execute the default implementation via Ruby's `super` keyword:

```
class Dog < Creature
  def fly
    puts "Woof!"
    super
  end
end

Dog.new.fly # Prints "Woof!"
           #         Sorry, I can't fly"
```

## Abstract Methods

Abstract methods are methods that can appear only in abstract classes. They contain only the signature of the method, without any implementation. For example, our `Creature` class can have two abstract methods: `Sleep` and `Wakeup`. Another method defines the flow of a single day:

```
public abstract class Creature
{
  public abstract void Sleep();
  public abstract void Wakeup();

  public void DayFlow()
  {
    Wakeup();
    Sleep();
  }
}
```

Now every creature has the same “day flow,” but its sleep and wakeup operations differ.

Implementing abstract methods in IronRuby is done just as you do with regular methods. The methods are defined with the same name of the abstract methods. Make sure that `super` is not available here, because the superclass doesn't have an implementation:

```
require "creature.dll"
class Human < Creature
  def wakeup
    puts "Good morning!"
  end
  def sleep
```

```

    puts "ZzzZzzzz"
  end
end

Human.new.DayFlow # Prints "Good morning!"
                  #      ZzzZzzzz"

```

## PARTIALLY IMPLEMENTING ABSTRACT CLASSES

If you don't implement all abstract methods, IronRuby still lets you construct the object, but an error will be raised when those methods are called.

### Regular Methods

Regular methods cannot be inherited. However, they can be hidden. In C#, you need to use the new keyword to give a new implementation to regular methods (also called *name hiding*). In IronRuby, the process is catchy because every CLR method has two names: the CLR one and the Ruby-like one. To entirely hide a method, you must implement the method twice (or alias the method to respond to the second name, too):

In C# (human.dll)

```

public class Human
{
    public void Eat()
    {
        Console.WriteLine("Going to a restaurant");
    }
}

```

In IronRuby

```

require "human.dll"
class Vegetarian < Human
  def eat
    puts "Going to vegetarian restaurant"
  end
  alias :Eat :eat
end
class Carnivore < Human
  def eat
    puts "Going to a fast food place"
  end
  alias :Eat :eat
end

```

```
Vegetarian.new.eat # Prints "Going to vegetarian restaurant"
Carnivore.new.Eat # Prints "Going to a fast food place"
```

If you hide the original implementation, it cannot be invoked directly anymore. If you do want to invoke it, you can use the `super` keyword or use the `clr_member` method, get the Method object of the CLR implementation, and invoke it:

```
class Vegetarian < Human
  def eat
    clr_member(:eat).call
    super
    puts "A vegetarian one"
  end
end
Vegetarian.new.eat # Prints "Going to a restaurant"
                  #      Going to a restaurant
                  #      A vegetarian one"
```

## Static Methods

In .NET languages, static methods are impossible to override. The original implementation can be hidden and replaced only by a new implementation.

Calling the base class implementation is impossible, as well, and the way to call it is by explicitly calling the base class name and method (`TheBaseClass.TheStaticMethod`).

The case with static methods in IronRuby is similar to what happens with regular methods. The original implementation is hidden by the new implementation. It is possible to hide CLR static methods by implementing them as class methods. In the following sample, I create a small-factory method for the `Human` class and change its implementation on the IronRuby Doctor class:

In C# (human.dll)

```
public class Human
{
  public static Human Create(string firstName)
  {
    Console.WriteLine("Creating a human");
    return new Human(firstName);
  }

  public Human(string firstName)
  {
  }
}
```

In IronRuby

```
require "human.dll"
class Doctor < Human
  def self.create(first_name)
    puts "Creating a doctor!"
    Doctor.new(first_name)
  end
end

Doctor.create("John") # Prints "Creating a doctor!"
                      # and returns a Doctor object
```

## Methods with Multiple Overloads

CLR methods, unlike Ruby ones, can be declared several times with different parameters. You learned about the mapping of such methods in Chapter 9. But, how do you override them?

Well, overriding them is easy, but doing so has consequences. When you override a multi-overloads method, you lose the overloads in IronRuby.

For example, take a look at the following C# class. It provides some overloads to the `Sleep` method, with different methods that supply different parameters for the sleep operation:

```
public class Human
{
  public virtual void Sleep()
  {
    Sleep(8);
  }
  public virtual void Sleep(int numberOfHours)
  {
    Sleep(numberOfHours, "bed");
  }
  public virtual void Sleep(int numberOfHours, string where)
  {
    Console.WriteLine("Sleeping {0} hours in {1}",
      numberOfHours, where);
  }
}
```

Now I implement a `RubyDeveloper` class that inherits from the `Human` class (assuming all Ruby developers are human). When this class is defined in IronRuby, all the `Human.Sleep` overloads are available:

```
require "human.dll"
class RubyDeveloper < Human
end
ruby_dev = RubyDeveloper.new
ruby_dev.sleep
# Output: "Sleeping 8 hours in bed"
ruby_dev.sleep(14)
# Output: "Sleeping 14 hours in bed"
ruby_dev.sleep(14, "IronBed")
# Output: "Sleeping 14 hours in IronBed"
```

If we want to override the parameterless `sleep` method overload on the `RubyDeveloper` class, so the default sleeping hours will be 14 rather than 8 (Ruby saves a lot of time, so Ruby developers can sleep more!), this is what we usually do:

```
class RubyDeveloper < Human
  def sleep
    super(14)
  end
end
```

Note the cost of this overridden method:

```
ruby_dev = RubyDeveloper.new
ruby_dev.sleep
# Output: "Sleeping 14 hours in bed"
ruby_dev.sleep(14) # Error! "wrong number of arguments"
ruby_dev.sleep(14, "IronBed") # Error! "wrong number of arguments"
```

## Sealed Methods

A sealed method is a method that implements an abstract or a virtual method and doesn't allow further inheritance from this method.

In CLR languages, it is impossible to inherit a sealed method, but it is possible to hide them. This is also what happens when you implement a sealed method in IronRuby: It hides the original sealed implementation.

Like regular method hiding, when you reimplement a sealed method, the CLR name and the Ruby name should both be reimplemented. Reimplementing only one name results in executing the original implementation when the other name is used.

In C# (`human.dll`)

```
public class Creature
{
    public virtual void Jump()
```

```
{
}
}
public class Human : Creature
{
    public override sealed void Jump()
    {
        Console.WriteLine("Jumping");
    }
}
```

In IronRuby

```
require "human"
class Athlete < Human
    def jump
        puts "Jumping very high"
    end
end
Athlete.new.jump # Prints "Jumping very high"
Athlete.new.Jump # Print "Jumping"
```

To work around this behavior, instead of writing the implementation twice, it is better to just alias the method names:

```
class Athlete < Human
    alias :Jump :jump
end
Athlete.new.jump # Prints "Jumping very high"
Athlete.new.Jump # Print "Jumping very high"
```

## Overriding Properties

CLR properties, like methods, can be overridden in the .NET Framework. Their behavior is identical to method overriding, with one major difference. Properties in IronRuby appear as two methods: a getter and a setter. To override a property, it is necessary to override both methods. It is also possible to override only the setter method or the getter method.

For example, consider the following C# class:

```
public class Human
{
    private string _fullName;
```

```

public virtual string FullName
{
    get
    {
        Console.WriteLine("Getting FullName");
        return _fullName;
    }
    set
    {
        Console.WriteLine("Setting FullName");
        _fullName = value;
    }
}
}

```

Using the `FullName` property in an IronRuby inheriting class writes “Setting FullName” and “Getting FullName” when accessing the property:

```

class Doctor < Human
end
doctor = Doctor.new
doctor.full_name = "John Doe" # Prints "Setting FullName"
doctor.full_name # Prints "Getting FullName"

```

To redefine the property, I override the getter and setter methods:

```

class Doctor < Human
    def full_name
        puts "IronRuby Getter"
        @full_name
    end
    def full_name=(value)
        puts "IronRuby Setter"
        @full_name = value
    end
end

doctor = Doctor.new
doctor.full_name = "John Doe" # Prints "IronRuby Setter"
doctor.full_name # Prints "IronRuby Getter"

```

Another way to override a property is by using accessors. To completely remove the CLR implementation and use the simple implementation of the accessor, the accessor name needs to be the same as the CLR accessor:

```
class Doctor < Human
  attr_accessor :full_name
end

doctor = Doctor.new
doctor.full_name = "John Doe" # Nothing is printed
doctor.full_name # Nothing is printed
```

CLR properties act the same as CLR methods with regard to access control variations. Refer back to the “Overriding Methods” section, earlier in this chapter, for a detailed explanation of each case.

## Overriding Events

CLR classes can contain virtual events. Therefore, they can be overridden by the class inheritors. The principles are similar to method overriding principles with a slight change. Like properties, events are split into two methods in IronRuby: `add_EventName` and `remove_EventName`.

For example, assume we have the next C# class:

```
public class Human
{
    public virtual event EventHandler ChangeMade;
    public void RaiseChangeMade()
    {
        if (ChangeMade != null)
        {
            ChangeMade(this, EventArgs.Empty);
        }
    }
}
```

In IronRuby, it is possible to override the event as follows:

```
class Doctor < Human
  def add_ChangeMade(handler)
    puts "Adding #{handler.inspect}"
    # Call the Human class "add event handler" implementation
    super
  end
  def remove_ChangeMade(handler)
    puts "Removing #{handler.inspect}"
    # Call the Human class "remove event handler" implementation
  end
end
```



```

    super
  end
end

```

Now adding an event handler from IronRuby to the Doctor class outputs a message as well:

```

d = Doctor.new
d.ChangeMade { puts "A change has been made" }
# Prints "Adding System.EventHandler"
d.raise_change_made
# Prints "A change has been made"

```

## Opening CLR Classes

When a CLR method is used in IronRuby, it instantly becomes a Ruby object. Ruby objects are dynamic, permissive, and open. Therefore, we can take a CLR class, for example, open it, and add more methods to it.

We do it exactly as we would have done it with pure Ruby classes. For example, the following code sample adds a `multiply` method to the CLR String class:

```

class System::String
  def multiply(amount)
    str = self
    (1..amount).each { str = str + self }
    str
  end
end

clr_string = System::String.new("IronRuby")
puts clr_string.multiply(3)
# Prints "IronRubyIronRubyIronRuby"

```

The change we made to the `System.String` class affects every `System.String` across the IronRuby application.

## Using Mixins

One of the most powerful features of the Ruby language is mixins (as covered in detail in Chapter 6). When opening CLR classes, it is possible to mix in modules to them, as well.

In the following code sample, I implement the `each` method for the `System.String` class and then mix in the `Enumerable` mixin:

```

class System::String
  include Enumerable

  def each
    (0...self.length).each { |i| yield self[i] }
  end
end

str = System::String.new("Test String")
str.max # = "T"
str.member?("s") # = true
str.sort # = %q{e g i n r s S t t T}

```

The `max` and `sort` methods are available because `System.Char` implements the `<=>` method when it is mapped to IronRuby.

## MIXIN CLR INTERFACES TO CLR CLASSES

As mentioned in Chapter 9, CLR interfaces are mapped to Ruby modules so that they can be used as mixins. This is correct only with regard to pure Ruby classes. It is impossible to mix CLR interfaces into CLR classes.

This behavior happens because mixing in a CLR interface to a CLR class means the class should inherit from the interface. IronRuby maps only CLR types; they do not become pure Ruby classes. Therefore, changing the class inheritance list after it's been created is not possible in static .NET languages. As a result, trying to perform this operation ends with an error.

## Opening the Object Class

IronRuby and the CLR share a single base object: `System.Object` (as discussed in Chapter 6).

This design makes it easy for us to add functionality to all objects in IronRuby, including CLR types (by opening the `Object` class).

For example, we can add a method that discovers whether the current object is pure Ruby:

```

class Object
  def self.is_pure_ruby?
    self.to_clr_type.nil?
  end
end

System::String.is_pure_ruby? # = false
Fixnum.is_pure_ruby? # = false (Fixnum is System.Int32)

```

```
class PureRubyClass; end
PureRubyClass.is_pure_ruby? # = true
```

## Opening Namespaces

Namespaces are mapped into Ruby modules. There is no problem with opening them and adding classes or methods to them.

The next example demonstrates how to add a new class to the `System.Data` namespace:

```
module System::Data
  class YamlDataSource
    # ... code ...
  end
end

yaml = System::Data::YamlDataSource.new
```

## Summary

In this chapter, you saw how the concepts of object-oriented programming in Ruby and the .NET Framework merge in IronRuby. You are now familiar with the main techniques (and some advanced ones), and you are now ready to learn how to take advantage of IronRuby's capabilities and the .NET world.

# PART IV

## IronRuby and the .NET World

### IN THIS PART

|            |  |     |
|------------|--|-----|
| CHAPTER 11 | Data Access                              | 259 |
| CHAPTER 12 | Windows Forms                            | 281 |
| CHAPTER 13 | Windows Presentation<br>Foundation (WPF) | 303 |
| CHAPTER 14 | Ruby on Rails                            | 331 |
| CHAPTER 15 | ASP.NET MVC                              | 363 |
| CHAPTER 16 | Silverlight                              | 401 |
| CHAPTER 17 | Unit Testing                             | 425 |
| CHAPTER 18 | Using IronRuby from<br>C#/VB.NET         | 459 |

*This page intentionally left blank*

# CHAPTER 11

## Data Access

Data access is a vital part of almost every application. After all, if an application stores data, data access is eventually necessary. This chapter focuses on data access in the context of accessing databases and using the obtained data. It is done via .NET Framework data access classes in IronRuby. The chapter guides you through using datasets, the `SqlClient` namespace, and the MySQL connector library in IronRuby.

### Hello, Data Access

Data access refers to the storage-related operations done in an application. It typically relates to a set of operations known as CRUD, which stands for Create Read Update Delete. Data access is incorrectly associated exclusively with database access. Although database access is a major component of data access, the term relates to every source of data, whether it is a database, file, web service, or public API.

In this chapter, I build a data access library that connects to a remote Microsoft SQL server. This library features a caching mechanism: When data is obtained from the SQL server, it is held in a local MySQL database. The local database data is cleaned after a specified amount of time.

As you see throughout this chapter, .NET Framework components make it easy to accomplish the complicated task of accessing databases. With the CLR interoperability support in IronRuby, these components are available for you to take advantage of.

### IN THIS CHAPTER

- ▶ Hello, Data Access
- ▶ Preparing Your Environment
- ▶ Contacting a SQL Server
- ▶ Contacting a MySQL Server
- ▶ Design Considerations
- ▶ The `CachedDataAccess` Class
- ▶ A Word About LINQ

## Preparing Your Environment

To run the code samples in this chapter, you need to download and install a few applications.

The first application is Microsoft SQL Server. Microsoft offers a free edition of the application, called SQL Server Express. You can download the latest edition from <http://www.microsoft.com/express/sql/default.aspx>.

The database I use in this chapter is the AdventureWorksLT sample SQL Server database. You can download it from <http://www.codeplex.com/MSFTDBProdSamples>.

The next application you need is MySQL. You can download a free edition of MySQL from <http://dev.mysql.com/downloads/mysql/5.1.html#downloads>.

You should also install GUI tools to make the administration process easier (<http://dev.mysql.com/downloads/gui-tools/5.0.html>).

The last package you need for this chapter is the MySQL connector. This is a .NET library for connecting the MySQL library. You can download it from <http://dev.mysql.com/downloads/connector/net/6.0.html>.

## Contacting a SQL Server

Contacting a SQL server is done via the `System.Data.SqlClient` namespace. This is the .NET way, and IronRuby is fully capable of taking advantage of it.

I build a class, named `SqlServerAccessor`, which handles the connection and execution of queries on the remote SQL server. The class is in a separate file named `sql.rb`.

### Loading the Needed Assemblies

To work with the `SqlClient` namespace, we have to load it first.

The `System.Data.SqlClient` namespace exists within the `System.Data` assembly. To load it, we can use the partial name only. (For more information about loading .NET assemblies, see Chapter 9, “.NET Interoperability Fundamentals.”)

```
require "System.Data"
```

When the assembly is loaded, we can start building our `SqlServerAccessor` class.

### Building the Class Structure

Building a class is simple and consists of defining its name. I’m also going to add a couple of `include` statements to the class to make developing inside the class easier. Writing an `include` statement inside the class scope, apart from being the way to mix in modules, acts like a `using` or `import` statement of C# and VB.Net; it brings the module content to the current scope:

```

class SqlServerAccessor
    include System::Data
    include System::Data::SqlClient
end

```

With these defined, instead of calling `System::Data::SqlClient::SqlConnection` every time, I can use `SqlConnection` directly.

## Building the Connection String

The first thing you must do before accessing a database is to define the connection string for it. The connection string is a semicolon-delimited string that defines the configuration of the connection to the database (server name, database name, whether the connection is trusted, and more).

Table 11.1 describes the common configuration properties for the SQL Server connection string.

TABLE 11.1 Common Configuration Keys for the SQL Server Connection String

| Key   | Description   | Values   |
|---|---|--|
| Data Source<br>or<br>Server<br>or<br>Address    | The network address of the SQL server, including the instance name if it exists.                            | Any string                                       |
| Initial Catalog<br>or<br>Database               | The name of the database.   | Any string                                       |
| Integrated Security<br>or<br>Trusted_Connection | When true, Windows authentication is used. When false, SQL Server authentication is used. Default is false. | True = true, yes, or sspi<br>False = false or no |
| User ID   | The SQL Server login account name. Used when integrated security is set to false.                           | Any string                                       |
| Password  | The password for the SQL Server login account. Used when integrated security is set to false.               | Any string                                       |

### Sample Connection Strings

The following are a few samples for SQL Server connection strings.

Here, we use integrated security for a database named `foo` that runs on a server called `sqlsrv`:

```
Data Source=sqlsrv;Database=foo;Integrated Security=true
```



In the following code, we use SQL Server security with a user named john with password doe. The server is sqlsrv, and database is called foo:

```
Server=sqlsrv;Initial Catalog=foo;Trusted_Connection=false;User
ID=john;Password=doe;
```

The following code shows a connection string for connecting a SQL Server Express instance on the local host, with a database named AdventureWorks using integrated security:

```
Integrated Security=True;Data Source=localhost\\sqlexpress;initial
➤catalog=AdventureWorks;
```

### Adding the Connection String to the Class

Now that you've learned how to build a connection string, we can add it as part of our class.

I add the connection string as an instance variable and initialize it within the constructor:

```
def initialize
  @connection_str = "Integrated Security=True;
                    Data Source=localhost\\sqlexpress;
                    Initial Catalog=AdventureWorks;"
end
```

## Opening a Connection to the SQL Server

The main task, after actually executing queries, is opening a connection to the database. After the connection has been opened, the queries can be made via it.

The class for SQL Server connectivity is `SqlConnection`. This class handles connections to the SQL Server and exposes events that help retrieving messages from the server, to know when the connection state has changed.

An in-depth look at this class is beyond the scope of this book. For more information about the class, search for `System.Data.SqlClient.SqlConnection` on the MSDN site.

In the `SQLAccessor` class, I add two methods that handle the server connection, one for opening a connection and one for closing it:

```
def open_connection
  # Open a connection only if it's not opened already
  if @connection.nil? || @connection.state != ConnectionState.open
    @connection = SqlConnection.new(@connection_str)
    @connection.open
  end
end

def close_connection
  @connection.close unless @connection.nil?
end
```

## Querying the Database

To use data stored in databases we must execute queries on them. A query is written in the SQL language, which except for small differences here and there has the same syntax on the various SQL servers (Microsoft SQL Server, MySQL, SQLite, and others).

The way to query a SQL Server in the `SqlClient` namespace is via the `SqlCommand` class. This class contains a SQL statement or a stored procedure call to a database and allows executing it.

To get the results in a convenient fashion, I use a `SqlDataAdapter` that fills a dataset with the command results.

The query is hard-coded in the method and gets the products whose name fully or partially matches the filter string:

```
def query(filter)
  # Make sure the connection is opened
  open_connection

  sql = <<SQL
    SELECT Name
    FROM SalesLT.Product
    WHERE Name like '%#{filter.sub("", " ")}%'
  SQL

  # Create a command
  command = SqlCommand.new(sql, @connection)
  command.command_type = CommandType.text
  # Create a data adapter for dataset filling
  adapter = SqlDataAdapter.new(command)
  dataset = DataSet.new
  # Fill the dataset with the data
  adapter.fill(dataset, "Product")

  # Return the filled dataset
  dataset
end
```

### TYPED DATASETS

As you might have noticed, in this chapter I use untyped datasets. I do so mainly to keep things simple.

Typed datasets are available in IronRuby, because they are regular .NET classes. However, you have to create them in Visual Studio, which means that until IronRuby is supported in Visual Studio you can create only typed-datasets written in C# or VB.Net.

To work around this problem, the typed dataset can be created via Visual Studio and saved to a .NET assembly. Then the assembly can be loaded in IronRuby, which makes the typed dataset available within the IronRuby code.

---

## Wrapping Up sql.rb

With all the code written in this section, we have created a class that connects to a SQL Server, executes a query on its data, and returns a dataset with results.

Listing 11.1 contains the whole content of the sql.rb file.

### LISTING 11.1 sql.rb File Content

---

```
require "System.Data"

class SqlServerAccessor
  include System::Data
  include System::Data::SqlClient

  def initialize
    @connection_str = "Integrated Security=True;
                      Data Source=localhost\\sqlexpress;
                      Initial Catalog=AdventureWorks;"
  end

  def query(filter)
    # Make sure the connection is opened
    open_connection

    sql = <<SQL
      SELECT Name
      FROM SalesLT.Product
      WHERE Name like '%#{filter.sub("'", "'')}%'
    SQL

    # Create a command
    command = SqlCommand.new(sql, @connection)
    command.command_type = CommandType.text
    # Create a data adapter for dataset filling
    adapter = SqlDataAdapter.new(command)
    dataset = DataSet.new
    # Fill the dataset with the data
    adapter.fill(dataset, "Product")
  end
end
```

```

    # Return the filled dataset
    dataset
end

def open_connection
  # Open a connection only if it's not opened already
  if @connection.nil? || @connection.state != ConnectionState.open
    @connection = SqlConnection.new(@connection_str)
    @connection.open
  end
end

def close_connection
  @connection.close unless @connection.nil?
end
end

```

---

## Using the SqlServerAccessor Class

Before we move on, let's try out our new class.

The next code uses our SQL Server class to search for helmets in the product table and presents the results. You can create an rb file out of it and run it or just execute it from the ir console:

```

require "sql.rb"
sql = SqlServerAccessor.new
data = sql.query("helmet")
for row in data.Tables["Product"].rows
  puts row["Name"]
end
sql.close_connection

```

Try it out and watch the results getting printed on the screen.

## Contacting a MySQL Server

The MySQL server instance will be used as a local database that I will deploy to the client's machine. As a result, our database instance should match the data we retrieve from the main server, the SQL Server. The class itself will be similar in its structure to the SQL Server class, too, because it needs to serve the same needs. The MySQL class will be stored in a separate file named `mysql.rb`.

## Preparing the MySQL Database

The first thing we need to do is to create a database on the MySQL server that includes that data we need.

To do that, I create a database instance with a single table named `Product`. This table contains an auto-incremented ID column, a `Name` column that holds the product name, and a `CreationDate` column that helps to determine whether this cached row is still relevant. (The requirements for the class state that the cache has a time limit.)

After you have installed MySQL on your local machine and configured its basic properties, proceed with these steps:

1. Go to Start > Run.
2. Type `<MySQL installation folder>\bin\mysql.exe` and press Enter.  
The installation folder should be something like `C:\Program Files\MySQL\MySQL Server 5.1`.
3. Insert the password of the root user you have specified during the Configuration Wizard.
4. Now the MySQL console is ready. The first task is to create a database. Type the following and press Enter:

```
create database adventureworks;
```

5. After we have a database, we need to create the table. Type the following SQL statement and press Enter:

```
CREATE TABLE 'adventureworks'. 'Product' (
  'ID' INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
  'Name' VARCHAR(255) NOT NULL,
  'CreationDate' DATETIME NOT NULL,
  PRIMARY KEY ('ID')
);
```

6. The last task is to create a user to query the database we've just created. First we create a user named John with the password Doe. Insert the next line and press Enter:

```
CREATE USER 'john' IDENTIFIED BY 'doe';
```

7. Now grant this user permission to query and delete rows from the database. Insert the following into the console and press Enter:

```
GRANT SELECT, INSERT, DELETE
ON adventureworks.*
TO 'john';
```

8. That's it. We're done. Type **quit** and press Enter to exit the console.

This process can also be done through a graphical user interface called MySQL Administrator, which is a part of the MySQL GUI Tools package.

## Loading the Assemblies

After you install the MySQL connector library, a few new files will be added to the Global Assembly Cache. The file we need is called `MySQL.Data`, and it is loaded as follows:

```
require "MySQL.Data",
      Version=6.0.4.0, Culture=neutral,
      PublicKeyToken=c5687fc88969c44d"
```

`MySQL.Data` uses enum values from the `System.Data` assembly, and therefore we need to require it, too:

```
require "System.Data"
```

## Building the Class Structure

After we have loaded the required assembly, we need to create a class to contain the needed operations for the MySQL server. The class will be called `MySQLAccessor`. I include the MySQL main namespace to make it easier to use its inner classes afterward:

```
class MySQLAccessor
  include System::Data
  include MySQL::Data::MySQLClient
end
```

## Building the Connection String

Following the connection string configuration keys mentioned in the previous section, we can build the connection string to the MySQL server. This time we cannot use integrated security, because MySQL does not support Windows users. (It is a multiplatform database.) Therefore, we have to define the username and password explicitly. The connection string will be as follows:

```
Server=localhost;Database=adventureworks;User ID=john;Password=doe
```

This connection string will be set to an instance variable on the class's `initialize` method:

```
def initialize
  @connection_str = "Server=localhost;Database=adventureworks;
                    User ID=john;Password=doe"
end
```

## Opening a Connection to the MySQL Server

The MySQL connector library has a similar object model to .NET's `SqlClient` library. This is the reason why the next methods are almost identical to the SQL Server class ones, with the small difference of using a different connection object. I create two methods, one for opening a connection and one for closing it. The name of the connection class is `MySQLConnection`, which might remind you of its SQL Server equivalent, `SqlConnection`:

```
def open_connection
  # Open a connection only if it's not opened already
  if @connection.nil? || @connection.state != ConnectionState.open
    @connection = MySQLConnection.new(@connection_str)
    @connection.open
  end
end

def close_connection
  @connection.close unless @connection.nil?
end
```

## Querying the Database

Querying the MySQL database is done using the `MySQLCommand` class. The table and column names are the same as in the SQL Server database, and therefore this method is almost identical to the SQL Server class:

```
def query(filter)
  # Make sure the connection is opened
  open_connection

  sql = <<SQL
    SELECT Name
    FROM Product
    WHERE Name like '%#{filter.sub("'", "'')}%'
  SQL

  # Create a command
  command = MySQLCommand.new(sql, @connection)
  command.command_type = CommandType.text
  # Create a data adapter for dataset filling
  adapter = MySQLDataAdapter.new(command)
  dataset = DataSet.new
  # Fill the dataset with the data
  adapter.fill(dataset, "Product")
end
```

```

    # Return the filled dataset
    dataset
end

```

## Inserting Records

The MySQL database is used as a local cache. Therefore, it should support insertion of new records. The next method does just that; it receives the row details and inserts them into the database:

```

def insert_record(product_name)
  # Make sure the connection is opened
  open_connection

  sql = "INSERT INTO Product(Name,CreationDate)
        VALUES('{product_name.sub("'", "'')}', NOW())"

  # Create a command
  command = MySqlCommand.new(sql, @connection)
  command.command_type = CommandType.text
  # Execute the delete command
  # This will also return the number of affected rows
  command.execute_non_query
end

```

This SQL statement doesn't return records, so the method returns only the number of affected rows.

## Deleting Records

A cache mechanism requirement is to delete old records. Hence, the MySQL class should support record removal. I add another method that deletes records older than the current time. As earlier, this SQL statement doesn't return records, so the method returns the number of affected rows:

```

def delete_old_records
  # Make sure the connection is opened
  open_connection

  sql = "DELETE FROM Product
        WHERE CreationDate < NOW()"
  # Create a command

```



```

command = MySqlCommand.new(sql, @connection)
command.command_type = CommandType.text
# Execute the command and delete the records
# This will also return the number of affected rows
command.execute_non_query
end

```

## Wrapping Up mysql.rb

The MySQLAccessor class is based on an open source .NET assembly and not on a built-in assembly like System.Data.

Listing 11.2 contains the whole class implementation that appears in the mysql.rb file.

### LISTING 11.2 mysql.rb File Content

---

```

require "System.Data"
require "MySQL.Data",
      Version=6.0.4.0, Culture=neutral,
      PublicKeyToken=c5687fc88969c44d"

class MySQLAccessor
  include System::Data
  include MySQL::Data::MySQLClient

  def initialize
    @connection_str = "Server=localhost;
                      Database=adventureworks;
                      User ID=john;Password=doe"
  end

  def query(filter)
    # Make sure the connection is opened
    open_connection

    sql = <<SQL
      SELECT Name
      FROM Product
      WHERE Name like '%#{filter.sub("'", "'')}%'
    SQL

    # Create a command
    command = MySqlCommand.new(sql, @connection)
    command.command_type = CommandType.text
    # Create a data adapter for dataset filling
    adapter = MySqlDataAdapter.new(command)

```

```
dataset = DataSet.new
# Fill the dataset with the data
adapter.fill(dataset, "Product")

# Return the filled dataset
dataset
end

def insert_record(product_name)
  # Make sure the connection is opened
  open_connection

  sql = "INSERT INTO Product(Name,CreationDate)
        VALUES('#{product_name.sub("'", "'')}', NOW())"

  # Create a command
  command = MySqlCommand.new(sql, @connection)
  command.command_type = CommandType.text
  # Execute the delete command
  # This will also return the number of affected rows
  command.execute_non_query
end

def delete_old_records
  # Make sure the connection is opened
  open_connection

  sql = "DELETE FROM Product
        WHERE CreationDate < NOW()"

  # Create a command
  command = MySqlCommand.new(sql, @connection)
  command.command_type = CommandType.text
  # Execute the delete command
  # This will also return the number of affected rows
  command.execute_non_query
end

def open_connection
  # Open a connection only if it's not opened already
  if @connection.nil? || @connection.state != ConnectionState.open
    @connection = MySqlConnection.new(@connection_str)
    @connection.open
  end
end
```

```

def close_connection
  @connection.close unless @connection.nil?
end
end

```

---

## Using the MySQLAccessor Class

We have now finished writing our MySQL class. Let's give it a try.

The next code uses the class to create, query, and delete the local MySQL database:

```

require "mysql.rb"
mysql = MySQLAccessor.new
# Create a records
mysql.insert_record("Iron helmet")
mysql.insert_record("Blue helmet")
# Query
data = mysql.query("helmet")
for row in data.Tables["Product"].rows
  puts row["Name"]
end
# Delete records
mysql.delete_old_records
mysql.close_connection

```

## Design Considerations

As a result of the similarity in the object model of the SQL Server classes and MySQL classes, the classes are very similar. We can take advantage of the Ruby language and save several lines of code.

I do that by creating a mixin module. This module contains the basic methods needed for a generic database server connection. The module assumes existence of four variables, as follows, that determine the type of server to communicate with:

- ▶ `@connection_str`: The connection string to the database server
- ▶ `@connection_class`: The class for connecting the server (like `SqlConnection`)
- ▶ `@command_class`: The class for creating a SQL command (like `SqlCommand`)
- ▶ `@adapter_class`: The class that is used to fill a dataset with the command results (like `SqlDataAdapter`)

When all these variables exist, the code can initialize the classes and use them.

The module contains a more general implementation of the query methods. Two methods are at hand: `execute_query`, which execute a given SQL statement and return a dataset with the results (good for SELECT statements); and `execute_non_query`, which execute a given SQL statement and return the number of affected rows (good for INSERT, UPDATE, and DELETE statements).

Listing 11.3 contains the module code.

---

LISTING 11.3 The `SqlAccessor` Module (`sql_accessor.rb`)

---

```
require "System.Data"

module SqlAccessor
  include System::Data

  def execute_query(sql, table_name)
    # Make sure the connection is opened
    open_connection

    # Create a command
    command = @command_class.new(sql, @connection)
    command.command_type = CommandType.text
    # Create a data adapter for dataset filling
    adapter = @adapter_class.new(command)
    dataset = DataSet.new
    # Fill the dataset with the data
    adapter.fill(dataset, table_name)

    # Return the filled dataset
    dataset
  end

  def execute_non_query(sql)
    # Make sure the connection is opened
    open_connection

    # Create a command
    command = @command_class.new(sql, @connection)
    command.command_type = CommandType.text
    # Execute the command
    command.execute_non_query
  end

  def open_connection
    # Open a connection only if it's not opened already
    if @connection.nil? || @connection.state != ConnectionState.open
```

```

        @connection = @connection_class.new(@connection_str)
        @connection.open
    end
end

def close_connection
    @connection.close unless @connection.nil?
end
end

```

---

Notice how the names of the classes were changed to the variables. This is another capability of the Ruby language. In static languages like C#, you can pass a type to a method and then call this type's constructor with a few lines of unattractive reflection code. In Ruby, this is very straightforward; the type can be passed around and be used just as if it were the type itself. This behavior makes the following line possible:

```
command = @command_class.new(sql, @connection)
```

Now that we have this module in hand, the `SqlServerAccessor` class size is greatly reduced, as you can see in Listing 11.4.

---

#### LISTING 11.4 The Modified `SqlServerAccessor` Class (`sql.rb`)

---

```

require "sql_accessor.rb"

class SqlServerAccessor
  include System::Data::SqlClient
  include SqlAccessor

  def initialize
    @connection_str = "Integrated Security=True;
                      Data Source=localhost\\sqlexpress;
                      Initial Catalog=AdventureWorks;"
    @connection_class = SqlConnection
    @command_class = SqlCommand
    @adapter_class = SqlDataAdapter
  end

  def query(filter)
    sql = <<SQL
      SELECT Name
      FROM SalesLT.Product
      WHERE Name like '%#{filter.sub("'", "'')}%'
    >>
  end
end

```

SQL

```
    execute_query(sql, "Product")
  end
end
```

---

The implementation is now reduced to a minimal amount of code. Only the necessary configuration variables are set, and the generation of the SQL statement is done locally on the class. The rest is done via the mixin module methods.

The `MySQLAccessor` class also benefits from the new mixin module, as shown in Listing 11.5.

---

LISTING 11.5 The Modified `MySQLAccessor` Class (`mysql.rb`)

---

```
require "MySQL.Data",
      :Version=6.0.4.0, :Culture=neutral,
      :PublicKeyToken=c5687fc88969c44d"
require "sql_accessor.rb"

class MySQLAccessor
  include MySQL::Data::MySQLClient
  include SqlAccessor

  def initialize
    @connection_str = "Server=localhost;
                      Database=adventureworks;
                      User ID=john;Password=doe"
    @connection_class = MySqlConnection
    @command_class = MySqlCommand
    @adapter_class = MySQLDataAdapter
  end

  def query(filter)
    sql = <<SQL
      SELECT Name
      FROM Product
      WHERE Name like '%#{filter.sub("'", "'')}%'
SQL

    execute_query(sql, "Product")
  end

  def insert_record(product_name)
    sql = "INSERT INTO Product(Name,CreationDate)
          VALUES( '#{product_name.sub("'", "'')}', NOW() )"
  end
end
```

```

        execute_non_query(sql)
    end

    def delete_old_records
        sql = "DELETE FROM Product
              WHERE CreationDate < NOW()"
        execute_non_query(sql)
    end
end

```

---

In the modified `MySQLAccessor` class, we can see that the connection and command handling has been moved to the `mixin` module, and the class remains to handle MySQL related code only.

## The CachedDataAccess Class

Now that we have the database access classes, we can use them in our cached data access class. The class contains a single method: `get_products`. The method first checks whether the local MySQL database contains the requested data. If not, the SQL Server is queried, and the results are cached in the local MySQL database:

```

# Require the accessor files
require "sql.rb"
require "mysql.rb"

class CachedDataAccess
    def get_products(name)
        # Try to get the data from the MySQL database
        mysql = MySQLAccessor.new
        data = mysql.query(name)

        if (data.tables["Product"].rows.count == 0)
            # Try to retrieve data from the remote SQL Server
            sql = SqlServerAccessor.new
            data = sql.query(name)
            # Add the returned rows to the local database
            for row in data.Tables["Product"].rows
                mysql.insert_record(row["Name"])
            end
        end

        # Return the result to the user
        data
    ensure

```

```
    # Close the connections
    mysql.close_connection unless mysql.nil?
    sql.close_connection unless sql.nil?
  end
end
```

Another task the cache class should do is to delete old records. For that, I create a thread that runs the `delete_old_records` method every 60 seconds:

```
def initialize
  Thread.new do
    # Loop endlessly
    loop {
      begin
        # Wait 60 seconds
        sleep(60)

        # Remove old records
        mysql = MySQLAccessor.new
        mysql.delete_old_records
      ensure
        # Close the connection
        mysql.close_connection unless mysql.nil?
      end
    }
  end
end
```

## Wrapping Up `cached_data_access.rb`

The preceding code blocks create our needed data access class. Listing 11.6 contains the entire class code.

LISTING 11.6 CachedDataAccess Class (`cached_data_access.rb`)

---

```
# Require the accessor files
require "sql.rb"
require "mysql.rb"

class CachedDataAccess
  def initialize
    Thread.new do
      # Loop endlessly
      loop {
        begin
```



```

        # Wait 60 seconds
        sleep(60)

        # Remove old records
        mysql = MySQLAccessor.new
        mysql.delete_old_records
    ensure
        # Close the connection
        mysql.close_connection unless mysql.nil?
    end
}
end
end

def get_products(name)
    # Try to get the data from the MySQL database
    mysql = MySQLAccessor.new
    data = mysql.query(name)

    if (data.tables["Product"].rows.count == 0)
        # Try to retrieve data from the remote SQL Server
        sql = SqlServerAccessor.new
        data = sql.query(name)
        # Add the returned rows to the local database
        for row in data.Tables["Product"].rows
            mysql.insert_record(row["Name"])
        end
    end

    # Return the result to the user
    data
ensure
    # Close the connections
    mysql.close_connection unless mysql.nil?
    sql.close_connection unless sql.nil?
end
end
end

```

---

## Using the CachedDataAccess Class

The class we created in this chapter exposes a single method only and is very easy to use.

Listing 11.7 uses the class and shows the user products that are related to his or her search string. The code exists when the user inputs q.

---

**LISTING 11.7** Using the `CachedDataAccess` Class (`main.rb`)

---

```
require "cached_data_access.rb"

cached_dal = CachedDataAccess.new

# Get initial input from the user
print "Insert search keyword: "
input = gets.chomp
while input != "q"
  # Search
  data = cached_dal.get_products(input)

  # Show results
  puts "Found #{data.Tables["Product"].rows.count} item(s)"
  for row in data.Tables["Product"].rows
    puts row["Name"]
  end

  # Get more input from the user
  puts
  print "Insert search keyword: "
  input = gets.chomp
end
```

---

To sum it up, we built the following files in our application:

- ▶ **sql\_accessor.rb**: The mixin module that is used by the SQL Server and MySQL classes to reduce code duplication.
- ▶ **sql.rb**: Contains the class responsible for accessing the SQL Server.
- ▶ **mysql.rb**: Contains the class that accesses the MySQL server.
- ▶ **cached\_data\_access.rb**: Contains the class that has the caching mechanism logic.
- ▶ **main.rb**: A sample use of the `CachedDataAccess` class.

## A Word About LINQ

LINQ (Language-Integrated Query) has become quite popular since it first appeared in .NET 3.0. You might wonder why only a word about it, then. Well, IronRuby doesn't support LINQ. The familiar syntax of LINQ is a DSL (Domain-Specific Language), which is internal to C# and VB.Net.

IronRuby in its first version does not support LINQ syntax. Even though Ruby's lambdas are similar to LINQ syntax, behind the scenes they are implemented entirely different and cannot (currently) be converted to each other.

LINQ methods, like `where`, do exist in IronRuby, but you must pass them an expression tree, which you have to build by yourself (a complicated task).

## Summary

In this chapter, you learned to access different databases using IronRuby. You learned how the use of built-in and custom .NET assemblies is identical, how to use datasets, and how to take advantage of the Ruby language to reduce code duplication.

Data access is a common practice that appears in almost every application. Therefore, you should become familiar with the available classes and possibilities so that you can leverage them during the development process.

# CHAPTER 12

## Windows Forms

**W**indows Forms, or WinForms for short, refers to the Windows graphical user interface (GUI) support of the .NET Framework. It consists of base UI classes and out-of-the-box controls (such as text box, grid, and various layout organization controls). WinForms is common and used extensively in Windows applications.

In this chapter, you learn how to use WinForms in IronRuby. You also learn how to build the initial user interface, using various UI controls, and how to respond to user actions. This chapter also covers the various ways you can create a UI (with or without a visual designer).

### Introduction

Since the early days of the .NET Framework, WinForms has been one of the major parts. It was a breath of fresh air after C++ MFC and made the process of developing Windows GUIs much easier than it was until then.

The basic component of every WinForm is the form. A form is an instance of the `Form` class, and it is generally a container of other controls with a few properties of its own. For example, a form can present Maximize, Minimize, and Close buttons. It also has a title, a size, and more.

Not only the form is a container, but every control in the form is a container as well. This allows complex UI components to be created like grids, trees, or tabular layouts. To reduce the complexity of code, the Windows Forms framework is designed using the composite design pattern. Therefore, everything within this framework has the same base class: `System.Windows.Control`. Even the `Form` object

### IN THIS CHAPTER

- Introduction
- Building the Chat Class
- Building the Chat Windows Form
- Writing the Execution Code

itself is a `Control`. Every control can contain other controls, and so eventually we get a hierarchy of controls that build up a whole form.

In this chapter, you learn the different aspects of WinForms development in IronRuby by creating a WinForms application. The application is a simple peer-to-peer chat application, for which Ruby libraries for connection between peers will be used.

## The Application Structure

The chat application is separated into three classes:

- ▶ `ChatForm` class: The WinForms form implementation, which contains the form and controls definitions
- ▶ `Chat` class: The logic class, which is responsible for the communication with the other chat window
- ▶ `ChatRunner` class: Prepares the environment for running a WinForms application and shows the form

## Building the Chat Class

The `Chat` class is responsible for the communication between two chat applications, sending and receiving chat messages.

To communicate between separate processes, maybe even on different machines, we need some kind of a communication class. The `TCPSocket` library is used for this purpose. You can choose any of several other options (for example, the `XmlRPC` library or .NET's web services).

To keep this example simple, it is assumed that both chat applications exist on the same machine. Changing this behavior is easy and requires passing the server name to this class.

## Requiring the Needed Assemblies

The first task for the `Chat` class is to require the standard library that handles TCP sockets. That library, as mentioned in Chapter 7, "The Standard Library," is `socket`:

```
require "socket"
```

## Initiating the Class

The `Chat` class receives one argument for its `initialize` method. This parameter represents the side of the chat communication. Every user should choose a different side, and according to the chosen value, the ports for sending and receiving messages are determined:

```
class Chat
  def initialize(side)
    if side == "A"
```

```

        @receive_port = 9988
        @send_port = 9989
      else
        @receive_port = 9989
        @send_port = 9988
      end
    end
  end
end

```

As you can see, I decided to use ports 9988 and 9989 for the application. This was an arbitrary pick and not a safe one: When choosing ports, make sure the ports are free and not already taken by other applications.

## Receiving Messages

The `listen` method is the one responsible for receiving chat messages from the other end. This is done using the `TCPServer` class. The `TCPServer` class opens the required port and waits for incoming connections. When a message is received, it is processed, and the server returns to wait for more connections.

If you come from the .NET world, you might consider events for the task at hand: When an incoming message is accepted, an event will be raised. In Ruby, there is no such object as an event. There are several other possibilities, such as the observer design pattern (the observable standard library can be used for that). In this method, the incoming message is passed to an associated code block using `yield`:

```

def listen
  server = TCPServer.new("localhost", @receive_port)
  # Wait for incoming messages
  while session = server.accept
    # Send the request to the associated code block
    yield session.gets
    # Close the current session
    session.close
  end
end

```

## Sending Messages

The `send` method receives a message argument, opens a connection to the outgoing port, and sends the message there. For that matter, I use the simple `TCPSocket` class. There is no need here for a continuous communication; the message is sent, and the operation is done:

```

def send(message)
  # Open the response socket
  s = TCPSocket.new("localhost", @send_port)

```

```

# Send the message
s.puts message
# Close the response socket
s.close
end

```

## Wrapping Up the Chat Class (chat.rb)

The Chat class is very much a general-purpose TCP communication class. It takes advantage of the basic operations needed in every TCP-driven application and can be easily transformed for other uses.

Listing 12.1 joins the section's code samples together into a single complete class.

LISTING 12.1 The Chat Class (chat.rb)

---

```

require "socket"
class Chat
  def initialize(side)
    if side == "A"
      @receive_port = 9988
      @send_port = 9989
    else
      @receive_port = 9989
      @send_port = 9988
    end
  end
end

def listen
  server = TCPServer.new("localhost", @receive_port)
  # Wait for incoming messages
  while session = server.accept
    # Send the request to the associated code block
    yield session.gets
    # Close the current session
    session.close
  end
end

def send(message)
  # Open the response socket
  s = TCPSocket.new("localhost", @send_port)
  # Send the message
  s.puts message
end

```

```
# Close the response socket
s.close
end
end
```

---

## Building the Chat Windows Form

The common way of building Windows Forms is via Visual Studio. Visual Studio offers a UI, called a *visual designer*, that features a way to build a form in an interface similar to the end result. It supports drag and drop of controls to the form, allowing developers to place them and set their properties without writing code. Behind the scenes, the visual designer generates code (C# or VB.Net) that creates the form and its controls with the settings that have been set during the design process.

Unfortunately, IronRuby lacks Visual Studio integration in its first version. Therefore, the visual designer cannot generate IronRuby code. As a result, the process of building a WinForm UI in IronRuby becomes a complicated and awkward task.

The way to create a WinForm in IronRuby is to just write the code that the visual designer would have written automatically if it were available.

### Loading the Needed Assemblies

To develop a WinForms application, two assemblies need to be loaded:

`System.Windows.Forms` and `System.Drawing`. The first is the main assembly and contains all the form-related classes and controls. The second is used mainly for several enum values or property classes (such as `Size` or `Point`).

These assemblies have reference files within the IronRuby Libs folder, so there's no need to state the strong name, only the partial one:

```
require "System.Windows.Forms"
require "System.Drawing"
```

Because it will be more convenient afterward, you should also include the namespaces:

```
include System::Windows::Forms
include System::Drawing
```

### Building the Class

The `ChatForm` class is a Windows Form class. Even though we can control a form from outside its scope, it is better to inherit from the `System.Windows.Forms.Form` class. This can give us more control over the form's internal state.



Therefore, the class can inherit from the `Form` class as follows:

```
class ChatForm < Form
  include System::Windows::Forms
  include System::Drawing
end
```

## Initializing the Form

The first task for this `Form` class is to initialize it. The `initialize` method code can call the methods that build the form in the right order—set form properties, create controls and register to control events:

```
def initialize
  # Set the form properties
  set_form
  # Create the controls
  create_controls
  # Register and response to control events
  register_events
end
```

The `set_form`, `create_controls` and `register_events` methods are discussed next in this chapter.

## SUPPRESSING LAYOUT EVENTS

When a form is built, several controls are added to it, and some events contain other controls themselves. Each control is then set with different properties to make it fit to the form layout.

The controls and the form are refreshed with almost every change like this. In a small form such as ours, we haven't noticed that. In bigger forms, however, the refreshes might result in a big performance impact.

However, the Windows Forms framework has a solution to that. Every control supports two methods: `SuspendLayout` and `ResumeLayout`. When you plan to change multiple properties of a specific control, it is recommended to call `SuspendLayout` at the beginning and `ResumeLayout` at the end. This way the control is drawn only at the end, with all the new properties applied simultaneously.

If the change, as in our case, is done on numerous controls on the form, it is more convenient to suspend and resume layout of the container control only. In our case, that would be the form itself: `self.SuspendLayout` and `self.ResumeLayout`.

---

## Setting the Form Properties

A Windows form has numerous properties. It is possible to control almost every aspect of the form, from its start position, through its resizing abilities, and to its background color or image.

A detailed explanation of the different form properties is beyond the scope of this book. For a complete reference, take a look at the MSDN website at <http://msdn.microsoft.com/en-us/library/system.windows.forms.form.aspx>.

Table 12.1 describes the commonly used properties.

TABLE 12.1 Commonly Used Form Properties

| Property Name            | Type                                 | Description   |
|--------------------------|--------------------------------------|---|
| ClientSize<br>or<br>Size | System.Drawing.Size                  | ClientSize is the size of the window without the borders and the title bar.<br>Size includes them.<br>It is enough to set only one of them.   |
| ControlBox               | System.Boolean                       | false hides the Minimize, Maximize, Close buttons strip at the top of the form.<br>True shows them (the default).   |
| MinimizeBox              | System.Boolean                       | false disables the Minimize button at the top of the form.<br>true shows it.  |
| MaximizeBox              | System.Boolean                       | false disables the Maximize button at the top of the form.<br>true shows it.  |
| FormBorderStyle          | System.Windows.Forms.FormBorderStyle | Defines the look and functionality of the form border. The available values are as follows:<br>None: No borders at all<br>FixedSingle: With borders and control box, no resize<br>Fixed3D: Like FixedSingle, but with a 3D border<br>FixedDialog: Like FixedSingle, but without a form icon<br>Sizable: Like FixedSingle, but allows resizing (FormBorderStyle default value)<br>FixedToolWindow: Like FixedDialog, but with smaller title bar and no Maximize and Minimize buttons<br>SizableToolWindow: Like FixedToolWindow, but allows resizing |

TABLE 12.1 Commonly Used Form Properties

| Property Name | Type                                       | Description  |
|---------------|--|--|
| StartPosition | System.Windows.Forms.<br>FormStartPosition | Defines the position of the form as it loads. The available values are as follows:<br>Manual: Defined manually using the Location property<br>CenterScreen: Positioned at the center of the screen<br>WindowsDefaultLocation: Positioned at the Windows default location (StartPosition default value)<br>WindowsDefaultBounds: Positioned at the Windows default location with a Windows default window size<br>CenterParent: Positioned at the center of the parent window |
| Text          | System.String                              | The title of the form.   |

For the chat application, I use only a few of those; I leave the rest at their default values. I set the form to be at a fixed size (no resizing), disable maximizing, and use the title “The IronRuby Chat”:

```
def set_form
  # Set the form size to 330 pixels width and 340 pixels height
  self.client_size = Size.new(330,340)
  # Do not allow resizing the window
  self.form_border_style = FormBorderStyle.fixed_single
  # Set the window title to "The IronRuby Chat"
  self.text = "The IronRuby Chat"
  # Disable the maximize button
  self.maximize_box = false
end
```

Note that the `self` keyword is used to set the form’s property values (like `self.text` and not just `text`). Properties must be approached that way because otherwise the property names will be treated as local variables and won’t affect the form object.

After these properties are set, the form looks like Figure 12.1.

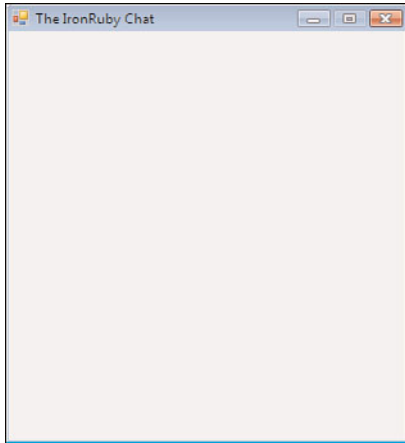


FIGURE 12.1 The application UI after the basic properties are set.

## Adding Controls

At this point, we have a form, but it is empty and isn't really handy. To make it user friendly, we should add controls to it.

Every Windows Forms control inherits from the `System.Windows.Forms.Control` class. This base class defines several methods and properties that are commonly used and thus are good to know. Table 12.2 describes these.

TABLE 12.2 Commonly Used Control Properties

| Property Name | Type                              | Description   |
|---------------|-----------------------------------|---|
| Location      | <code>System.Drawing.Point</code> | The location of the control on the form. <code>Point.new(0,0)</code> is the top-left point on the form.                         |
| Size          | <code>System.Drawing.Size</code>  | The size of the control. The first parameter is the width, and the second is the height: <code>Size.new(width, height)</code> . |
| Text          | <code>System.String</code>        | The text of the control. For example, a text box contains the text inside, and a check box displays the text nearby.            |
| Enabled       | <code>System.Boolean</code>       | <code>true</code> makes the control accessible for the user. <code>false</code> makes it inaccessible.                          |

Every control adds additional properties that are related to its own behavior (as you'll see in this section).

Let's start adding controls to the form. I start with the connection part on the top of the form. This part lets the user choose his side of the chat and connect to the chat. To build that we need three controls: a label to display the title of the combo box, a combo box that lets the user choose her side of the chat, and a button to connect to the chat.

The first task is to create these controls and save them to instance variables:

```
@side_combo = ComboBox.new
@side_label = Label.new
@connect_button = Button.new
```

Now that we have the controls, we can set their properties. The first control is the `label` control. I set its text and location:

```
# Set the location 15 pixels from the top and from the left borders
@side_label.location = Point.new(15, 15)
# Set the text to "Chat Side"
@side_label.text = "Chat Side"
```

The `label` control also contain a property for setting the font family, size, and style (`Font`) and one for changing the text color (`ForeColor`).

The next control to set is the combo box. Here, along with its location, I need to add the available values:

```
# Set the location 70 pixels from the left and 10 from the top
@side_combo.location = Point.new(70, 10)
# Set its size to 170 pixels width and 20 pixel height
@side_combo.size = Size.new(170,20)
# Set its style. DropDownList doesn't allow custom input to the list.
@side_combo.drop_down_style = ComboBoxStyle.DropDownList;
# Add the available items
@side_combo.items.add "Side One"
@side_combo.items.add "Side Two"
# Set the first item as the default one
@side_combo.selected_index = 0
```

The control in this part is the button. Its properties are similar to the other controls:

```
# Set the button location to 245 pixels from the left and 10 from the top
@connect_button.location = Point.new(245, 10)
# Set the size to 72 pixels width and 22 pixels height
@connect_button.size = Size.new(72, 22)
# Set the text to "Connect"
@connect_button.text = "Connect"
```

After this part is done, the form looks like Figure 12.2.

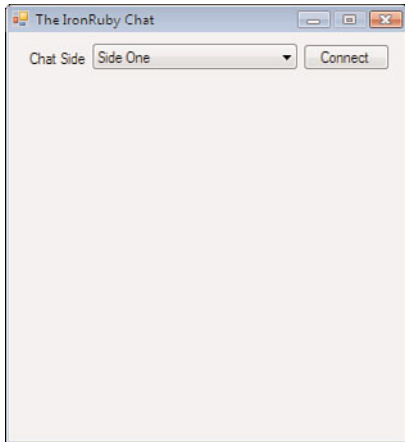


FIGURE 12.2 The application UI after the connection part is added.

The rest of the form is the conversation part. It contains the conversation text, a text box to write a new message, and a button to send the new message:

```
@conversation_textbox = TextBox.new
@message_textbox = TextBox.new
@send_button = Button.new
```

The conversation text will be presented inside a read-only text box. This way the user can see the conversation text but cannot modify it:

```
# Set the textbox location to 15 pixels from the left and 40 from the top
@conversation_textbox.location = Point.new(15, 40)
# Set the textbox size to 300 pixel width and 230 pixels height
@conversation_textbox.size = Size.new(300, 230)
# Enable multiline content
@conversation_textbox.multiline = true
# Make the textbox read-only
@conversation_textbox.read_only = true
```

The next control is the message text box:

```
# Set the textbox location to 15 pixels from the left and 275 from the top
@message_textbox.location = Point.new(15, 275)
```

```
# Set the textbox size to 230 pixels wide and 45 pixels tall
@message_textbox.size = Size.new(230, 45)
# Enable multiline content
@message_textbox.multiline = true
```

The last control on the chat form is the button to send messages. The button is located right next to the message text box and is called Send:

```
# Set the button location to 250 pixels from the left and 275 from the top
@send_button.location = Point.new(250, 275)
# Set the button size to 64 pixels width and 45 pixels height
@send_button.size = Size.new(64, 45)
# Set the button text to "Send"
@send_button.text = "Send"
```

After all this is written, the form design is done. Figure 12.3 shows how this should look.

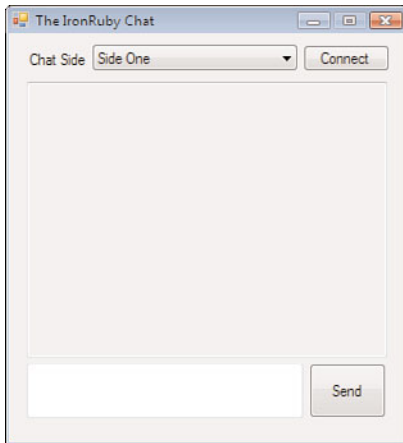


FIGURE 12.3 The finished application UI.

As the last task to complete the form design, we need to actually add the controls to the form. This is done by adding the controls to the `controls` list of the form:

```
self.controls.add @side_combo
self.controls.add @side_label
self.controls.add @connect_button
self.controls.add @conversation_textbox
self.controls.add @message_textbox
self.controls.add @send_button
```

Note that without this last step of adding the controls to the `controls` collection, the controls will not be presented on the form at all.

## Adding Functionality

Currently, we have the form ready with all the needed controls inside. The problem is that the form is not interactive; the buttons do not react to the user actions. The task now is to give life to these buttons and make the form react to user actions. This is where the previous class, `Chat`, comes into play.

Our goal is to make the `Connect` button start a listening thread by using the `listen` method and make the `Send` button execute the `send` method with the relevant parameters.

The WinForms framework is based on the observer pattern for handling user actions. In C# or VB.Net, this comes more naturally via events: Every UI control contains several events that notify subscribers of different actions. The subscribers then take the necessary actions according to the raised event. For example, buttons support the `Click` event, which is raised when the button is clicked; text boxes support the `TextChanged` event, which is raised after the user changes its content; and combo boxes feature the `SelectedIndexChanged` event, which notifies of a new value selection.

### CLR EVENTS IN IRONRUBY

In IronRuby, we do not have events, but as discussed in Chapter 9, “.NET Interoperability Fundamentals,” the way to use and respond to .NET events is easy and straightforward.

Before we start handling events, we need to add a method we will need afterward. This method adds text to the conversation text box. It gets two parameters. The first, `from_me`, indicates whether the message is written by me or by the other chat member. The second parameter, `message`, is the message itself. The method generates the needed text and appends it to the text box:

```
def print_message(from_me, message)
  # Add user identification text
  @conversation_textbox.text += if from_me then "Me: " else "Friend: " end
  # Add the message and a new line afterwards
  @conversation_textbox.text += message + "\r\n"
end
```

After we have this helper method available, we can move forward and handle the first user action (clicking the `Connect` button). When the user clicks the `Connect` button, we need to create a new `Chat` class instance and pass it the user-selected chat side. *Connect* means



that we can start receiving messages from the other chat side, so we will also need to create a listening thread that prints messages once they arrive:

```
@connect_button.click {
    # Get the user selected chat side
    if @side_combo.selected_index == 0 then side = "A" else side = "B" end
    # Create a new chat class instance
    @chat = Chat.new(side)
    # Start a listening thread. Once a message arrives, print it
    Thread.new {@chat.listen { |m| print_message false, m }}
}
```

The next user action to handle is clicking the Send button. When the user clicks Send, there are three tasks to do (add the message to the conversation text box, send the message to the other chat side, and clear the message text box):

```
@send_button.click {
    # Check if the connection has been made
    if @chat.nil? then
        # Connection hasn't been made, show an error
        MessageBox.show("Please connect first.")
    else
        # Add the message to the conversation textbox
        print_message true, @message_textbox.text
        # Send the message to the other side
        @chat.send @message_textbox.text
        # Clear the textbox content
        @message_textbox.clear
    end
}
```

Notice the beginning of this last code block. we start by validating that a connection has been made. If the validation fails, an error is presented to the user using the `MessageBox` class. The `MessageBox` class, which is a part of the `System.Windows.Forms` namespace, is a very handy class. It features only the `show` method that shows a message box with a given message at the center of the screen. The method features numerous different parameters, such as the box title, icon, buttons, and more. You should familiarize yourself with the `MessageBox` class; you'll find it handy in various situations.

For a detailed description of the `MessageBox` class, take a look at the MSDN site at <http://msdn.microsoft.com/en-us/library/system.windows.forms.messagebox.aspx>.

## EVENTS PARAMETERS

In the examples in this chapter, I haven't used the parameters that are passed to the events. Events in the .NET Framework usually send two arguments: sender and args. The sender parameter contains the object that has raised the event, and args contains the information passed with the event (if any).

UI events are no exception. I could write the above event-handling code with these parameters:

```
@send_button.click { |sender, args| ... }
```

I didn't need this information for this example, but you might find them useful in your application. To find out which information is passed with the event, look for the event in the MSDN class and look at the args variable type.

## Using the Visual Studio Visual Designer

The process of designing WinForms without a visual designer isn't easy. However, we can still use the visual designer for a kick start.

To do so, we just create a new Windows Forms application in Visual Studio and design the form as we need while using everything that Visual Studio features. Afterward, we have two options: convert the designer code to IronRuby or save the form in a C#/VB.Net assembly and use it from IronRuby.

### Converting the Designer Code to IronRuby

When we finish designing our form, on the Solution Explorer the form file contains subfiles. One of them ends with .designer.cs or .designer.vb. Figure 12.4 shows the file we're looking for. (We're looking for it because it contains the designer-generated code.)

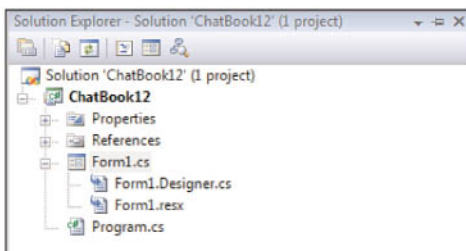


FIGURE 12.4 The Solution Explorer file tree.

All we have to do now is to take the C# or VB.Net code and convert it to Ruby code.

When doing so, follow these guidelines:

- ▶ Controls should be saved to instance variables. They do not need to be declared outside the method (like C# requires for instance variables). For example, `this.textBox1` should be converted to `@textBox1`.
- ▶ Initialization should be done the Ruby way. For example
 

```
this.textBox1 = new System.Windows.Forms.TextBox();
```

 should be converted to
 

```
@textBox1 = System::Windows::Forms::TextBox.new
```
- ▶ Explicit calls to namespaces should be done the Ruby way. For example, `System.Windows.Forms` should be converted to `System::Windows::Forms`.
- ▶ Form property and method calls should use the `self` keyword. For example, `this.ClientSize` or `this.SuspendLayout` should be converted to `self.ClientSize` or `self.SuspendLayout`. Be careful not to change `this.ClientSize` to `@ClientSize` or to `client_size` (`self.client_size` is okay) because they will not work and might even result in an exception.

### Using the Form from a .NET Assembly

Another way to skip the task of placing and setting control and form properties is by using the form from a .NET assembly. On the one hand, it makes your application not entirely Ruby; on the other hand, however, it saves you a lot of effort.

To do so, you just use Visual Studio as a visual designer, as follows:

1. Create a new Windows Forms application in Visual Studio.
2. Add controls and design one or multiple forms. Do not handle events or write any code; this will be done via IronRuby.
3. Delete the file `program.cs` or `program.vb` from the project. This file contains the code that runs the application, which we will write by ourselves in IronRuby.
4. Compile to an assembly. Let's assume that the name of the assembly is `ChatForm.dll`.
5. From IronRuby, load the assembly: `require "ChatForm.dll"`.
6. Initialize the form in IronRuby, handle the different events, and write any code that is needed for the application. No need for the `set_form` and `create_controls` methods from the `ChatForm` class introduced in this chapter.

Note that this time you don't create a form, and therefore you don't need to create a `Form` class that inherits from `System.Windows.Forms.Form`. The `Form` class has already been generated in the assembly, and all you have to do now is use the generated `Form` class.

## Wrapping Up the ChatForm Class

In this section, we built the ChatForm class, set the needed control properties, and handled user actions.

Listing 12.2 contains the entire code of the ChatForm class.

LISTING 12.2 The ChatForm Class (chat\_form.rb)

---

```
require "System.Windows.Forms"
require "System.Drawing"
require "chat"

include System::Windows::Forms
include System::Drawing

class ChatForm < Form

  def initialize
    # Set the form properties
    set_form
    # Create the controls
    create_controls
    # Register and response to control events
    register_events
  end

  def print_message(from_me, message)
    # Add user identification text
    @conversation_textbox.text += if from_me then "Me: " else "Friend: " end
    # Add the message and a new line afterwards
    @conversation_textbox.text += message + "\r\n"
  end

  def register_events
    @connect_button.click {
      # Get the user selected chat side
      if @side_combo.selected_index == 0 then side = "A" else side = "B" end
      # Create a new chat class instance
      @chat = Chat.new(side)
      # Start a listening thread. Once a message arrives, print it
      Thread.new { @chat.listen { |m| print_message false, m } }
    }
    @send_button.click {
      # Check if the connection has been made
```

```

    if @chat.nil? then
      # Connection hasn't been made, show an error
      MessageBox.show("Please connect first.")
    else
      # Add the message to the conversation textbox
      print_message true, @message_textbox.text
      # Send the message to the other side
      @chat.send @message_textbox.text
      # Clear the textbox content
      @message_textbox.clear
    end
  }
end

def set_form
  # Set the form size to 330 pixels width and 340 pixels height
  self.client_size = Size.new(330,340)
  # Do not allow resizing the window
  self.form_border_style = FormBorderStyle.fixed_single
  # Set the window title to "The IronRuby Chat"
  self.text = "The IronRuby Chat"
  # Disable the maximize button
  self.maximize_box = false
end

def create_controls
  # init controls
  @side_combo = ComboBox.new
  @side_label = Label.new
  @connect_button = Button.new
  @conversation_textbox = TextBox.new
  @message_textbox = TextBox.new
  @send_button = Button.new

  # Set control properties

  # Set the location 70 pixels from the left and 10 from the top
  @side_combo.location = Point.new(70, 10)
  # Set its size to 170 pixels width and 20 pixel height
  @side_combo.size = Size.new(170,20)
  # Set its style. DropDownList doesn't allow custom input to the list.
  @side_combo.drop_down_style = ComboBoxStyle.DropDownList;
  # Add the available items
  @side_combo.items.add "Side One"
  @side_combo.items.add "Side Two"

```

```
# Set the first item as the default one
@side_combo.selected_index = 0
# Set the location 15 pixels from the top and from the left borders
@side_label.location = Point.new(15, 15)
# Set the text to "Chat Side"
@side_label.text = "Chat Side"
# Set the button location to 245 pixels from the left and 10 from the top
@connect_button.location = Point.new(245, 10)
# Set the size to 72 pixels width and 22 pixels height
@connect_button.size = Size.new(72, 22)
# Set the text to "Connect"
@connect_button.text = "Connect"
# Set the textbox location to 15 pixels from the left and 40 from the top
@conversation_textbox.location = Point.new(15, 40)
# Set the textbox size to 300 pixel width and 230 pixels height
@conversation_textbox.size = Size.new(300, 230)
# Enable multiline content
@conversation_textbox.multiline = true
# Make the textbox read-only
@conversation_textbox.read_only = true
# Set the textbox location to 15 pixels from the left and 275 from the top
@message_textbox.location = Point.new(15, 275)
# Set the textbox size to 230 pixels wide and 45 pixels tall
@message_textbox.size = Size.new(230, 45)
# Enable multiline content
@message_textbox.multiline = true
# Set the button location to 250 pixels from the left and 275 from the top
@send_button.location = Point.new(250, 275)
# Set the button size to 64 pixels width and 45 pixels height
@send_button.size = Size.new(64, 45)
# Set the button text to "Send"
@send_button.text = "Send"

# Add the controls to the form
self.controls.add @side_combo
self.controls.add @side_label
self.controls.add @connect_button
self.controls.add @conversation_textbox
self.controls.add @message_textbox
self.controls.add @send_button
end
end
```

---

## Writing the Execution Code

Now that we have almost everything ready, the last task left is to write code that will show the form.

This is a tricky part. The `Form` class has a `show` method, and it seems logical that this method will open and show the dialog. However, this is not quite correct. For an application to run as a Windows form, there are a few configuration calls to execute when the application starts.

These configuration methods are a part of the `System.Windows.Forms.Application` class. This is a static class (no instance is needed) with various different methods; most of them are related to handling the Windows message loop.

Two method calls should be done right after the application has started. These are calls to `Application.EnableVisualStyles` and `Application.SetCompatibleTextRenderingDefault`. The first makes the application use visual styles. Your application can run without this call, but it looks better with it. The second is also an optional but recommended call. It provides compatibility for a previous text-rendering technique used in .NET 1.0. If there is no obvious reason to do otherwise, this method should be passed with `false` as its argument (tells the controls to use the new text-rendering technique).

The last and most important call is the `run` method. To show the first form of the application (the next ones can be shown using the `show` method), the `run` method must be used. The `run` method starts the application message loop and shows the form passed as an argument.

To gather all of these operations, we create a single file named `main.rb`, as shown in Listing 12.3. Running this file shows the form.

LISTING 12.3 The Application Executing Code (`main.rb`)

---

```
require "System.Windows.Forms"
require "System.Drawing"
require "chat_form"

include System::Windows::Forms

# Enable visual styles
Application.EnableVisualStyles()
# Make controls use the recent text rendering technique
Application.SetCompatibleTextRenderingDefault(false)
# Create the form and show it
Application.Run(ChatForm.new);
```

---

Eventually, if we execute `main.rb` twice to open two chat windows, choose different sides, and click `Connect`, we can start the chat session and transfer messages between the chat windows, as shown in figure 12.5.

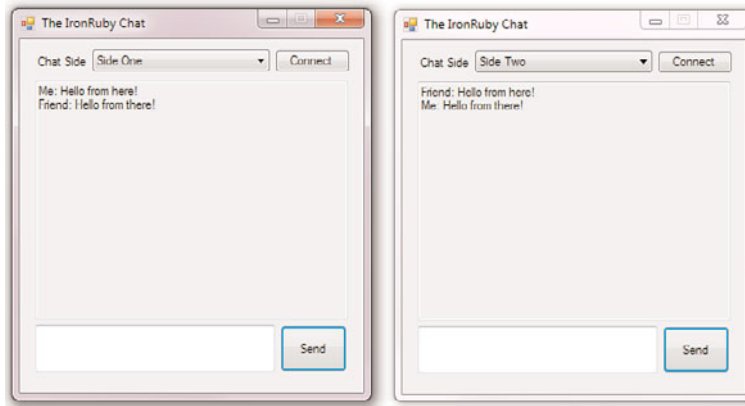


FIGURE 12.5 A sample chat session between two IronRuby-driven chat windows.

## Summary

In this chapter, we built a Windows Forms application with a user-friendly interface. You have learned the basic of the framework: the different controls, their properties, how to initiate them, and how to respond to user actions. You have also been introduced to a couple of workarounds to deal with the lack of a WinForms visual designer in IronRuby.

Windows Forms is an extensively used framework. It is easy to use and understand, and because it's been around awhile, it has numerous control suites that can make your application look very attractive.

However, a new presentation framework is available that offers better graphics capabilities and smarter designer/developer separation. This framework is called Windows Presentation Foundation (WPF) and is the subject of the next chapter.



*This page intentionally left blank*

## CHAPTER 13

# Windows Presentation Foundation (WPF)

**W**indows Presentation Foundation (WPF) is a graphical framework for creating rich user interfaces. Evolved from the WinForms framework, it provides a similar functionality (for example, input controls, data binding, and handling of user actions).

This is where the similarity ends. WPF is richer, and more mature, and makes complicated tasks turn to a matter of minutes. It's like WinForms on steroids.

## Hello, WPF

WPF is a presentation framework that was first introduced in .NET Framework 3.0. Remember that IronRuby can run on .NET 2.0 SP1, so to use WPF you need to install .NET 3.0 or later.

With WPF, several tasks that were really complicated on the WinForms framework are made much easier. For example, creating animations and image effects is a matter of seconds. However, with great power comes more complexity. You see throughout the chapter that mastering WPF is a bit harder than WinForms; it takes more than just coding and positioning components on a form.

But, one step at a time.

To create our first IronRuby-driven WPF application, we need to require the WPF assemblies. These are the `PresentationFramework` and `PresentationCore` assemblies:

```
require "PresentationFramework",  
       "Version=3.0.0.0",  
       "Culture=neutral",
```

## IN THIS CHAPTER

- ▶ Hello, WPF
- ▶ XAML
- ▶ IronRuby and WPF Fundamentals
- ▶ Windows
- ▶ Layout Controls
- ▶ Graphics and Animations
- ▶ Data Binding
- ▶ REPL

```

        PublicKeyToken=31bf3856ad364e35"
require "PresentationCore",
        Version=3.0.0.0,
        Culture=neutral,
        PublicKeyToken=31bf3856ad364e35"

```

These assemblies have references on the IronRuby Libs folder, so it is possible to require them with their partial name, too:

```

require "PresentationFramework"
require "PresentationCore"

```

Now that we have WPF at our fingertips, we can write our first WPF application:

```

require "PresentationFramework"
require "PresentationCore"

w = System::Windows::Window.new
t = System::Windows::Controls::TextBlock.new
t.text = "Hello, WPF!"
w.content = t

app = System::Windows::Application.new
app.run w

```

This code is pretty straightforward. After requiring the WPF assemblies, I created a window. To show text inside the window, I added a `TextBlock` element and set a welcome text to it. Then I set the `TextBlock` as the content of the window.

To run the WPF window, it is necessary to create the WPF application first and then run it. This is what is done on the last two lines.

There is another way to show the window via the `Application` class that some might find more appealing:

```

def app_start(sender, args)
  w = System::Windows::Window.new
  t = System::Windows::Controls::TextBlock.new
  t.text = "Hello, WPF!"
  w.content = t

  w.show
end

app = System::Windows::Application.new
app.startup.add method(:app_start)
app.run

```

The difference here is that we don't pass the window to the `Application.run` method, but we subscribe to the `startup` event and do our initialization there.

These code blocks generate the window shown in Figure 13.1.

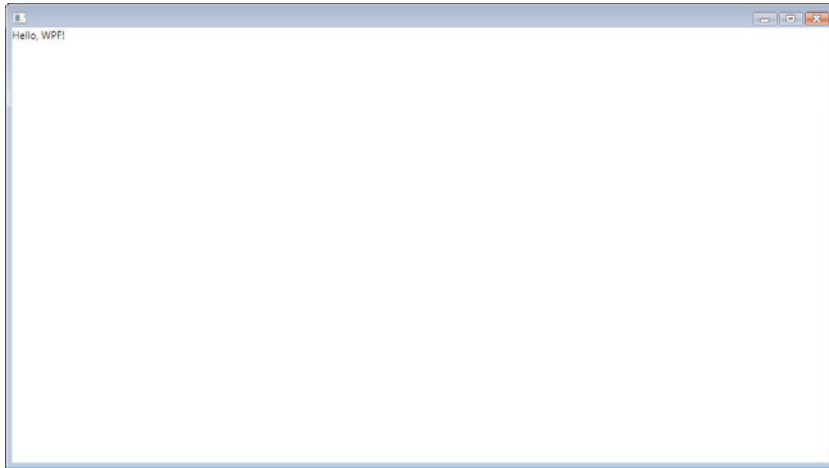


FIGURE 13.1 The first IronRuby-driven WPF window.

This window wraps up our first IronRuby-driven WPF application.

## XAML

On our Hello, WPF application, we have created the UI entirely via code. This might work for the first or second application but not for bigger complex applications.

XAML, which stand for eXtensible Application Markup Language, is an XML-based language used as the UI description language. The idea is brilliant—providing a full separation between UI and code. No more writing locations and sizes of controls in code; this can be done entirely in a convenient XML format.

For example, take the Hello, WPF application. Its equivalent XAML string is as follows:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <TextBlock Text="Hello, WPF!" />
</Window>
```

The XAML language is very powerful. Apart from being able to describe UI elements and their properties, it can hold resources and reference them.

For example, we could use a color name as a resource and use it on several different elements:

```

<Window
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Window.Resources>
    <SolidColorBrush Color="Fuchsia" x:Key="MyColor" />
  </Window.Resources>
  <StackPanel>
    <TextBlock x:Name="msg" Text="Hello, WPF!" Foreground="{StaticResource
MyColor}" />
    <Ellipse Width="100" Height="100" Fill="{StaticResource MyColor}" />
  </StackPanel>
</Window>

```

This XAML code ends up as the window shown in Figure 13.2.

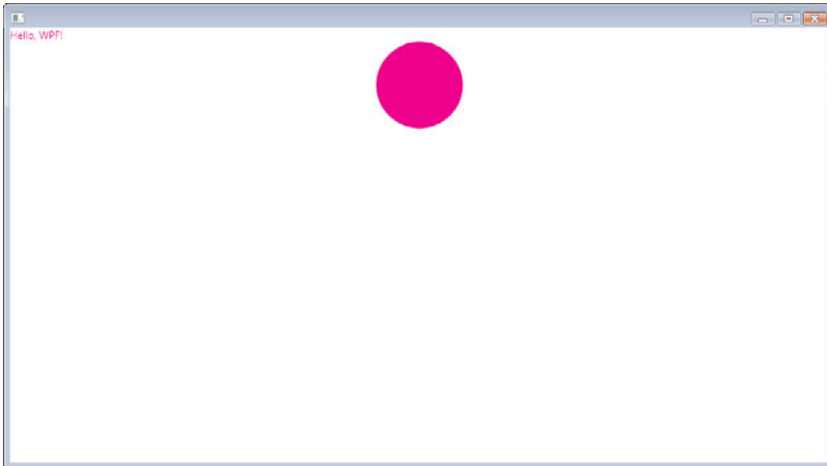


FIGURE 13.2 A WPF Window generated by XAML.

Static resources are discussed later in the chapter in the “Data Binding” section.

## XAML AND WPF

It is important to notice that XAML is an XML language. This means that XAML doesn’t have to be used solely by the WPF framework.

Any framework, even not a .NET one, that can interpret the XAML language can use it.

## Namespaces

XAML samples in this section contain namespace declaration (`xmlns` and `xmlns:x`), and you might wonder what they are for:

```
<Window
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
</Window>
```

The first namespace (`xmlns`) makes the elements correlate to .NET's WPF controls. Just like C# or VB.Net has namespaces to uniquely identify `System.Window.Forms.Control` and `System.Web.UI.Control`, XAML has these namespaces to uniquely identify WPF elements.

The second namespace (`xmlns:x`) is used for more WPF-specific elements and attributes. For example, naming objects is done via it: `x:Name`. Naming an element is done for a more convenient use via code, which makes it not a pure XAML property. This is why it appears within the second namespace and not the first one.

## IronRuby and WPF Fundamentals

To work with WPF via IronRuby, there are a few simple tasks to master. When you know these, you can accomplish tasks with ease with WPF and IronRuby.

### Running XAML

WPF is fully integrated with the XAML language. In current .NET static language, you can even use XAML windows as classes and have a direct connection to their content.

IronRuby, however, does not support this in version 1.0. We need to load the XAML file manually and access content in a fully dynamic way.

To load a XAML file, we can use the `parse` method of the `System::Windows::Markup::XamlReader` class. If the XAML appears on a file, we read it into a variable first and then use the `parse` method.

The `parse` method returns the root WPF object of the XAML file (`Window`, `NavigationWindow`, or `Page`, for example):

```
require "PresentationFramework"
require "PresentationCore"
xaml = File.open('d:/app/window1.xaml', "r").read
@root = System::Windows::Markup::XamlReader.parse(xaml)
```

Now that we have the root object, we can use the `Application` class `run` method or, if a WPF application is already running, we can use the `show` or `show_dialog` methods (if the root object is a window):

```
app = System::Windows::Application.new
app.run @root
# or
@root.show
```

## Retrieving WPF Elements

When you work with XAML, you do not have all the elements as objects within the code. This situation can easily be solved.

Every framework element provides the `find_name` method that searches for an element that matches the given name and returns it.

This means that if we have the root element (the window, for instance), we can find and access every element within it. For example, to find an element named `msg` within the root element, we write the following code:

```
@root.find_name("msg")
```

We can even enhance that and implement the `method_missing` method so that it locates elements for us:

```
def method_missing(name)
  super if @root.nil?
  control = @root.find_name(name)
  if control.nil? then super else control end
end
```

With that implemented, accessing `msg` is much more natural:

```
msg.text = "Hello from IronRuby!"
```

## Event Handling

Responding to user actions is one of the essentials of every client application. Every WPF element contains several events that will be raised under certain circumstances.

Registering to WPF events is done like every other CLR event, by using `add`, or with a code block. For example, if we have a button on the XAML code: `<Button x:Name="btn">Say Hello</Button>`

We can register to its `click` event:

```
def say_hello(sender, args)
  System::Windows::MessageBox.show("Hello!")
end
win.find_name("btn").click.add method(:say_hello)
# or
win.find_name("btn").click += method(:say_hello)
# or
win.find_name("btn").click do |s,e|
  System::Windows::MessageBox.show("Hello!")
end
```

WPF events bubble up until someone handles them. Sometimes we may want to stop the event bubbling. For example, we may want to disable closing the form, even by the Alt+F4 combination. To do that, we can catch that keyboard click, handle it, and stop the event from bubbling up (which eventually closes the form). We can do that by the arguments object that is passed with every WPF element event. This object is a `RoutedEventArgs` object. It contains a property named `Handled`, which is the key to our solution. If we set this one to true, it stops the event bubbling.

If you write a custom component or subscribe to an event from multiple locations, it is a good practice to check the `Handled` property before starting and not to proceed when it equals true.

## Windows

WPF features a few control containers that are used for displaying the UI in slightly different approaches.

If you are a Windows Forms developer, the following looks familiar to you, but don't you despair, WPF has a surprising new window type, `NavigationWindow`, which is described at the end of this section.

### WINDOW CONTENT

WPF windows can contain only a single item. There is no `Controls` collection or a similar concept. As weird as it sounds, it makes sense. In the WPF framework, these control containers are designed to provide the surface and not the layout mechanism. For example, you might want to place controls one after another or place them in fixed positions. This will be done via one of the several layout controls available (described later on the "Layout Controls" section) and not by the window itself.

## Window

A `Window` is a resizable control container that provides, except from the controls surface, a border and a title bar that contains an icon, a textual title, and control boxes (minimize, maximize, and close).

The following XAML code contains the most common `Window` attributes as well as Table 13.1:

```
<Window
    WindowStyle="SingleBorderWindow"
    ResizeMode="CanResize"
    WindowState="Normal"
    WindowStartupLocation="CenterScreen"
    Icon="windowIcon.ico"
    Title="Wpf Window"
    Width="500"
    Height="500"/>
```



TABLE 13.1   Common Attributes of Window

| Name                  | Description                                     | Value   |
|-----------------------|---|---|
| WindowStyle           | Controls the window border and title bar look   | SingleBorderWindow<br>ThreeDBorderWindow<br>ToolWindow<br>None  |
| ResizeMode            | Controls resizing and minimizing capabilities   | CanResize<br>CanResizeWithGrip<br>CanMinimize<br>NoResize   |
| WindowState           | Controls the state of the window as it opens    | Normal<br>Maximized<br>Minimized  |
| WindowStartupLocation | Controls the position of the window as it opens | Manual<br>CenterOwner<br>CenterScreen   |
| Icon                  | Controls the icon of the window                 | A path to the icon file. Can be a file system path or URI.  |
| Title                 | Sets the text on the title bar                  | A string.   |
| Width and Height      | Controls the size of the window                 | A double value followed by px (for pixels), in (for inches), cm (for centimeters) or pt (for points).<br>Default is pixels.<br>Can be set to Auto to enable automatic resizing. |

**WindowStyle**

The WindowStyle attribute has four possible values. Each value makes the window look slightly different. Table 13.2 describes the possible values.

TABLE 13.2   WindowStyle Values

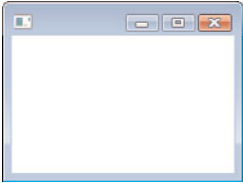
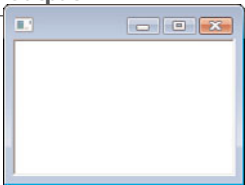
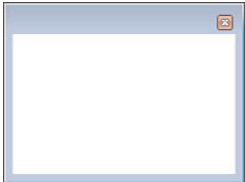

| Value              | Description                             | Output   |
|--------------------|---|--|
| SingleBorderWindow | Includes a border and a full title bar. |  |

TABLE 13.2 WindowStyle Values

| Value              | Description   | Output   |
|--------------------|---|--|
| ThreeDBorderWindow | Includes a 3D border and a full title bar.  |  |
| ToolWindow         | Includes a border and a title bar with no icon and no minimize/maximize boxes.                                    |  |
| None               | Does not include a border or a title bar at all.<br>A small border is added only when the window allows resizing. |  |

**ResizeMode**  
ResizeMode controls the resizing capabilities of the window. Unlike WinForms, this also controls the minimize/maximize boxes when needed. Table 13.3 describes the possible values.

TABLE 13.3 ResizeMode Values

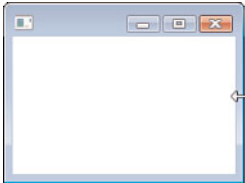
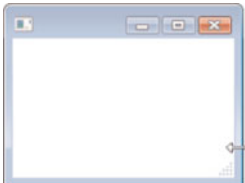
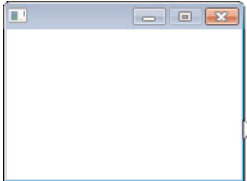
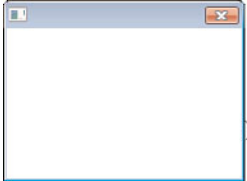
| Value             | Description   | Output   |
|-------------------|---|--|
| CanResize         | Default value. Enables resizing.                                    |  |
| CanResizeWithGrip | Enables resizing and adds a resize grip to the bottom-right corner. |  |

TABLE 13.3   ResizeMode Values

| Value       | Description  | Output   |
|-------------|--|--|
| CanMinimize | Disables resizing and maximizing the window. Only minimizing is enabled.   |  |
| NoResize    | Disables resizing. The maximize and minimize boxes become hidden.<br><br>Use this value to remove the border from a window with None as its WindowStyle. |  |

**Showing the Window**

There are three ways to display a window. The first one is via the `Application.run` method. By passing a window object to it, the window will be presented to the user and will act as the main window of the application. Note that you cannot use the `run` method more than once.

The second way is by executing the `show` method of the `Window` class.

The final way of to show a window is by using the `show_dialog` method of the `Window` class. This method, like the `show` method, will present the window immediately to the user. However, if you use the `show_dialog` method, the window becomes a *modal* window, which means that the user cannot access the open window (or other control container) until it is closed. Showing a window in modal mode also makes the code that opened it wait until it is closed. Modal windows are primarily used for dialog boxes for very specific tasks (choosing files, setting properties, and so on). There is no real difference in the way we build the window itself. The difference is only in the concept and goal of the window.

**Passing Data Between Windows**

Sometimes a need arises to pass data between two windows (or more). For example, one window might contain user input that the other window needs for its operations.

Achieving that is simple. We have code, so there is no problem with inheriting from the `Window` class and adding the needed attributes.

Listing 13.1 contains two windows: `main_window` and `dialog_box`. `main_window` opens the `dialog_box` window and takes the needed information from it after it is closed.

LISTING 13.1   Passing Data Between Windows

```
require "PresentationFramework"
require "PresentationCore"
```

```
main_window = System::Windows::Markup::XamlReader.parse <<XAML
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Width="400" Height="200">
  <StackPanel>
    <Button x:Name="button">Click</Button>
  </StackPanel>
</Window>
XAML
```

```
dialog_box = System::Windows::Markup::XamlReader.parse <<XAML
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Width="300" Height="150">
  <StackPanel
    HorizontalAlignment="Center" VerticalAlignment="Center">
    <TextBlock>What would you like to do?</TextBlock>
    <RadioButton x:Name="sleep">Sleep</RadioButton>
    <RadioButton x:Name="jump">Jump</RadioButton>
    <Button x:Name="ok_button">OK</Button>
  </StackPanel>
</Window>
XAML
```

```
main_window.find_name("button").click {
  # Show the dialog
  dialog_box.show_dialog
  # The code will reach the next condition only
  # after the dialog_box window is closed
  if dialog_box.find_name("sleep").is_checked
    System::Windows::MessageBox.show "Go to bed!"
  else
    System::Windows::MessageBox.show "Jump jump jump!"
  end
}

dialog_box.find_name("ok_button").click {
  dialog_box.close
}
```

```
app = System::Windows::Application.new
app.run main_window
```

---

With the preceding code, our application looks like Figure 13.3.

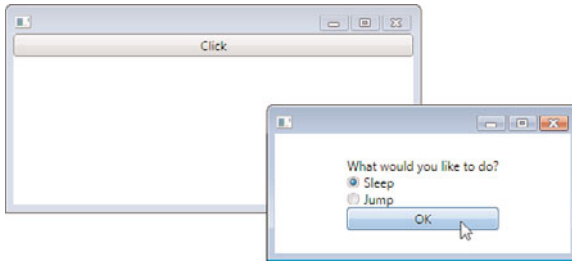


FIGURE 13.3 Passing data between windows.

When the user clicks OK in this situation, a “Go to bed!” message pops up.

## Navigation Window

Windows are a nice concept, but multiple windows that pop up every time is passé. With the fast rise of the Internet and rich web applications, users are less tolerant to Windows applications that open various windows on a regular basis.

This is where `NavigationWindow` saves the day. The concept is simple: You do not open different windows for the user; instead, you display the new content on the same window. The user then can navigate back and forward between the windows.

In addition to that, `NavigationWindow` has a built-in support for displaying web pages.

A blank navigation window is almost identical to a regular window. The addition on navigation windows is a navigation panel at the top of the window (which, of course, can be removed), as shown in Figure 13.4.

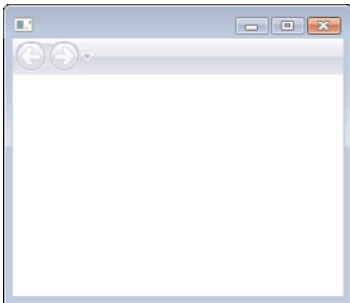


FIGURE 13.4 A WPF navigation window.

### NavigationWindow Properties and Methods

`NavigationWindow` supports all properties and methods that regular windows do. (It actually inherits from the `Window` class.)

Apart from it, `NavigationWindow` adds some more unique properties and methods. The most common are `ShowsNavigationUI` and `Source`:

- ▶ `ShowsNavigationUI` is a Boolean property (supports `true` and `false` values) that indicates whether the navigation panel should be shown or hidden.
- ▶ `Source` is a string property that can contain a URI to a website, a file, or a resource.

For example, the following XAML code shows the navigation panel and sets the source to `http://www.sampublishing.com`:

```
<NavigationWindow
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  ShowsNavigationUI="true"
  Source="http://www.sampublishing.com"/>
```

This XAML code is translated to the window shown in Figure 13.5.

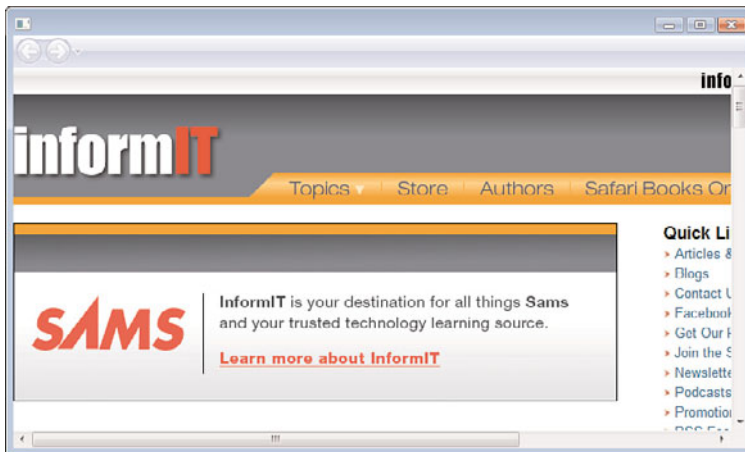


FIGURE 13.5 Using the navigation window to visit Internet sites.

As well as navigation-related properties, `NavigationWindow` has unique related methods:

- ▶ `navigate` changes the current content to the given object or URI.
- ▶ `refresh` reloads the current page.

## Content

Up until now, with regular windows, we got used to a certain format of an XAML file: the window declaration and then the controls it contains between its tags.

This is not the case with navigation windows. Navigation windows are lone wolves, and hence it is impossible to add controls between the `NavigationWindow` tags.

To use WPF controls as `NavigationWindow` content, we must declare them in a different XAML file (or object). The control container that is used for this purpose is `Page`.

A `Page` contains controls and can be contained within a `NavigationWindow`. As a control designed to be used inside navigation windows, the `Page` class has some attributes for controlling their parent window:

- ▶ `Title` sets the page title. This is not the title that the window will display. The place where you can see it is on the navigation list (the small arrow near the navigation buttons).
- ▶ `WindowHeight`, `WindowWidth`, and `WindowTitle` set the host window properties.
- ▶ `KeepAlive`, when set to `true`, keeps the page in memory after the user has navigated away. When the user returns, the page isn't rerendered.

Listings 13.2 and 13.3 contain two pages that the user can navigate between.

---

#### LISTING 13.2 First Navigation Page (page1.xaml)

---

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  WindowTitle="Page 1">
  <StackPanel>
    <TextBlock>This is page 1</TextBlock>
    <TextBlock>
      <Hyperlink NavigateUri="page2.xaml">
        Go to page 2!
      </Hyperlink>
    </TextBlock>
  </StackPanel>
</Page>
```

---



---

#### LISTING 13.3 Second Navigation Page (page2.xaml)

---

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  WindowTitle="Page 2">
  <StackPanel>
    <TextBlock>This is page 2</TextBlock>
```

```
<Button Command="NavigationCommands.BrowseBack">
    Back to page 1</Button>
</StackPanel>
</Page>
```

---

To make the application use these pages, just change the Source property of the navigation window to page1.xaml file URL:

```
<NavigationWindow
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    ShowsNavigationUI="true"
    Source="file:///C:/WPFApplication/page1.xaml"/>
```

## Layout Controls

Now that you know the different window types of the WPF framework, let's continue and explore the layout possibilities. As mentioned previously, windows supply the surface, and layout controls supply the layout mechanism. The different layout controls provide, each in its own special way, a layout organization mechanism.

The available layout controls are `StackPanel`, `WrapPanel`, `DockPanel`, `Grid`, and `ScrollViewer`.

There is no problem with combining several different layout controls. For example, a `Grid` can contain a `WrapPanel` inside one of its cells to provide wrapping capabilities to it.

### StackPanel

A `StackPanel` is a layout control that lets you arrange controls in a stack form (one on top of the other or side by side).

It is good for simple and small sections more than for big and complex ones.

For instance, the following XAML sample demonstrates a sample stack panel content:

```
<StackPanel>
    <TextBlock>Hello there</TextBlock>
    <Button>Click me</Button>
    <RadioButton>Check here</RadioButton>
    <Image Source="file:///c:/flower.jpg"/>
</StackPanel>
```



Figure 13.6 shows how this looks.

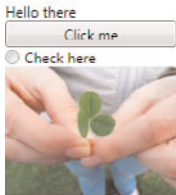


FIGURE 13.6 Using the StackPanel layout.

## PREVENTING ELEMENT STRETCHING

As you might have noticed, elements within the stack panel automatically stretch to fill its entire width. For example, look at the button in Figure 13.6. Sometimes this behavior might suit our needs, but most of the times it just looks weird.

We can prevent this from happening if we want to. To do so, just set the `HorizontalAlignment` property. For example, if we change the button definition as follows:

```
<Button HorizontalAlignment="Left">Click me</Button>
```

The button returns to its original size, as shown in Figure 13.7.

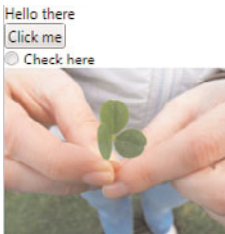


FIGURE 13.7 Using `HorizontalAlignment` property to prevent element stretching.

`StackPanel` supports a horizontal orientation, too, which places the UI elements side by side:

```
<StackPanel Orientation="Horizontal">
```

This changes the look of the panel from a vertical strip to a horizontal one, as shown in Figure 13.8.

## WrapPanel

`WrapPanel` is similar to `StackPanel` as they both place their child elements in vertical or horizontal orientations. The difference is that `WrapPanel` creates another column (in vertical orientation) or starts a new line (in horizontal orientation) when it runs out of space.

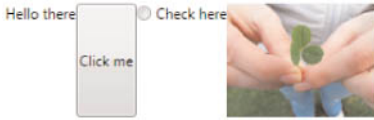


FIGURE 13.8 The StackPanel layout in horizontal orientation.

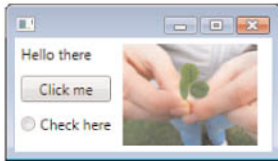


FIGURE 13.9 Using the WrapPanel layout.

For example, look how using WrapPanel in the following sample forces the image to move to a new column because there is not enough room for it on the first one:

```
<Window
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Width="230" Height="130">
  <WrapPanel Orientation="Vertical">
    <TextBlock Margin="5">Hello there</TextBlock>
    <Button Margin="5">Click me</Button>
    <RadioButton Margin="5">Check here</RadioButton>
    <Image Margin="5" Source="file:///d:/flower.jpg"/>
  </WrapPanel>
</Window>
```

## Grid

A grid provides a table layout mechanism. It enables you to organize the content in rows and columns. After defining the number and properties of the rows and columns, you can position elements within the grid cells.

### POSITIONING ELEMENTS WITHIN CELLS

Although multiple elements can be positioned within a single cell (stacked one on top of the other), you might want to consider also using panels to organize the cell content. (Nesting panels within panels is a great way to organize the whole and its parts.)

The following XAML sample contains a grid of two columns and three rows. Notice how the elements are positioned in the right cell (`Grid.Column` and `Grid.Row`) and the ability to span one cell over multiple columns or rows (`Grid.RowSpan` and `Grid.ColumnSpan`):

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="75" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>

  <Label Grid.Column="0" Grid.Row="0" Margin="5">Name:</Label>
  <TextBox Grid.Column="1" Grid.Row="0" Margin="5" />
  <CheckBox Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="2"
    Margin="5">Loves IronRuby</CheckBox>
  <Button Grid.Column="1" Grid.Row="2" Margin="5">
    Send</Button>
</Grid>
```

Figure 13.10 shows how this looks.

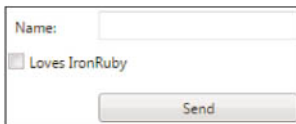


FIGURE 13.10 Using the Grid layout.

## Canvas

So far, we've been talking about panels that are relative to the window size. Canvas is different: Its element positioning agenda is the absolute one. Every element is positioned in a specific position that does not change according to canvas size changes.

The following sample sets the elements in fixed positions (`Canvas.Top`, `Canvas.Bottom`, `Canvas.Left`, and `Canvas.Right`):

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Canvas>
    <Button Canvas.Top="10" Canvas.Left="15">
      Click me!</Button>
```

```
<Button Canvas.Top="40" Canvas.Left="45">  
    I'm the one to click!</Button>  
<Button Canvas.Bottom="20" Canvas.Right="10">  
    No! Click me!</Button>  
</Canvas>  
</Window>
```

The top button (and the others according to their positions) will always be 10 pixels from the top and 15 from the left, no matter whether there is enough room.

The preceding XAML content generates the window shown in Figure 13.11.

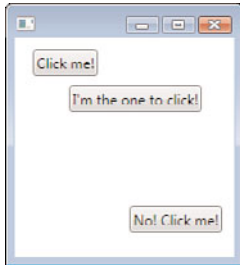


FIGURE 13.11 Using the Canvas layout.

## More Panels

WPF offers more panel variations, but these are beyond the scope of this book. These panels are `DockPanel`, `ScrollViewer`, and `ViewBox`.

You can read more about them on the MSDN website:

- ▶ `DockPanel`: <http://msdn.microsoft.com/en-us/library/system.windows.controls.dockpanel.aspx>
- ▶ `ScrollViewer`: <http://msdn.microsoft.com/en-us/library/system.windows.controls.scrollviewer.aspx>
- ▶ `ViewBox`: <http://msdn.microsoft.com/en-us/library/system.windows.controls.viewbox.aspx>

## Graphics and Animations

WPF is a great framework offering excellent UI flexibility and options. However, up to now I haven't described anything major that isn't possible with the WinForms framework.

The graphics capabilities of WPF are some of its "killer features" that have been much more complicated to achieve before.

## Shapes

There are various different shapes in the WPF framework, which together make it possible to build every shape out there. Table 13.4 briefly describes each of the available shapes.

TABLE 13.4   WPF Shapes

| Name      | Description  |
|-----------|--|
| Rectangle | Draws rectangles.  |
| Ellipse   | Draws ellipse shapes (circles and others).   |
| Line      | Draws a single line that starts at one point and ends at another. A line doesn't have to be a horizontal one.          |
| Polyline  | Instead of a line between two points, a polyline is a line that can be drawn between multiple points.                  |
| Polygon   | Just like polyline but creates a closed shape.   |
| Path      | The most complex shape and provides a large number of capabilities; can combine several shapes, draw curves, and more. |

For example, creating a drawing of a home with circular windows is simple:

```
<Canvas>
  <Rectangle Canvas.Left="88" Canvas.Top="80" Height="70"
    Width="90" Stroke="Black" />
  <Polyline Canvas.Left="88" Canvas.Top="50"
    Stroke="Black" Points="0,30 45,0 90,30"/>
  <Ellipse Canvas.Left="95" Canvas.Top="85" Height="22"
    Width="23" Stroke="Black" />
  <Ellipse Canvas.Left="145" Canvas.Top="85" Height="22"
    Width="23" Stroke="Black" />
  <Rectangle Canvas.Left="119" Canvas.Top="119"
    Height="31" Width="26" Stroke="Black"/>
</Canvas>
```

The XAML code results in the drawing shown in Figure 13.12.

## Brushes

Every element that is drawn in a WPF window uses a brush for drawing itself. WPF provides several brushes that offer different ways to paint a control, as described in Table 13.5.

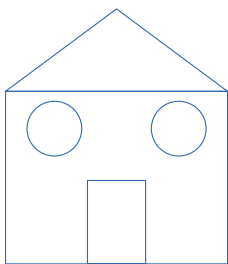








FIGURE 13.12 Using shapes.

TABLE 13.5 WPF Brushes

| Name                | Description  | Sample  |
|---------------------|--|---|
| SolidColorBrush     | Draws using a single color   |    |
| LinearGradientBrush | Draws with a transformation between two colors                             |    |
| RadialGradientBrush | Start with one color and transforms to a second one in an elliptical shape |    |
| ImageBrush          | Draws using a given picture  |   |
| DrawingBrush        | Draws using a given drawing (defined with WPF shapes)                      |  |
| VisualBrush         | Draws using a given WPF element (every element acceptable)                 |  |

The screenshots in the preceding table are made from a rectangle using the different brushes for filling itself. To give you an idea how easy this is, look at the code responsible for the linear gradient sample:

```
<Rectangle Height="100" Width="100" Margin="20">
  <Rectangle.Fill>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
```

```

        <GradientStop Color="Yellow" Offset="0" />
        <GradientStop Color="Green" Offset="1" />
    </LinearGradientBrush>
</Rectangle.Fill>
</Rectangle>

```

It is important to note that brushes can be used wherever a color can be used (for example, even for filling text).

## Animations

WPF has built-in extensive support for animation. The animation possibilities are extensive: size changing, rotation, transition effects, moving in paths, and more.

Every WPF element can be animated. We are not limited here to video clips or GIF animations; we can actually animate a text box, for example.

The simplest and most commonly used animation is the one that changes a specific property. For example, to animate the size of a control, the `Height` and `Width` properties should be changed, or to animate the background color, the `Background` property should be changed, and so on.

The following sample XAML code generates a `TextBlock` that animates the font size from 1 pixel to 60 pixels after the `TextBlock` is loaded:

```

<TextBlock Name="txt" FontSize="1px">Hello
  <TextBlock.Triggers>
    <EventTrigger RoutedEvent="TextBlock.Loaded">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation
            Storyboard.TargetName="txt"
            Storyboard.TargetProperty="FontSize"
            From="1.0" To="60.0" Duration="0:0:2"
            AutoReverse="True" RepeatBehavior="Forever" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </TextBlock.Triggers>
</TextBlock>

```

The important element for us here is the `DoubleAnimation` element. `DoubleAnimation` can manipulate values of type `double`. On this example, `FontSize` is a `double` value that makes `DoubleAnimation` a perfect match.

There are more animation elements, one for every type. Each of them support a `From-To` set of values: `ByteAnimation`, `ColorAnimation`, `DecimalAnimation`, `Int16Animation`, `Int32Animation`, `Int64Animation`, `Point3DAnimation`, `PointAnimation`,

QuaternionAnimation, RectAnimation, Rotation3DAnimation, SingleAnimation, SizeAnimation, ThicknessAnimation, Vector3DAnimation, and VectorAnimation.

You might sometimes want to create animations via code. As in XAML, you must define a `DoubleAnimation` object and connect it to the property you want to animate. The following code is the equivalent to the preceding XAML code:

```
include System::Windows::Media::Animation
include System::Windows::Controls
include System::Windows

anim = DoubleAnimation.new
anim.from = 1.0
anim.to = 60.0
anim.duration = Duration.new(System::TimeSpan.from_seconds(2))
anim.repeat_behavior = RepeatBehavior.forever
anim.auto_reverse = true
win.find_name("txt").BeginAnimation(
    TextBlock.font_size_property, anim)
```

The WPF framework includes even more animation capabilities. You can define animations on a timeline, use key frames, and move elements around on a given path. These are not described here, but you can read about them on the MSDN website.

## Data Binding

Applications are mostly about displaying data. Layout arrangements, special graphics, and animations are there to enhance the users' experience while they are obtaining data.

Data binding in WPF is an integral part of every single property. You can bind text to a given data, which is the common case, and you can also bind the color, font size, or position to the data.

### Binding to Static Data

Sometimes there are static values that you would like properties to bind to (for example, if you want a specific color to appear on the application or a special text message).

Static data is called a resource. The location of the resources is within the `Windows.resources` XAML element. Every resource has a special name to reference afterward. This name is defined on the `x:Name` attribute.

The next XAML code contains color and string resources. Note that the extra namespace allows using CLR types:

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```



```

xmlns:s="clr-namespace:System;assembly=mscorlib">
<Window.Resources>
  <SolidColorBrush x:Key="olive_color" Color="Olive" />
  <s:String x:Key="ironruby_text">IronRuby!!!</s:String>
</Window.Resources>
</Window>

```

You can leverage these resources straight from the XAML code or from the IronRuby code.

From XAML code, we would use the binding special value format. The format consists of curly brackets with the `StaticResource` keyword and the resource name right afterwards: `{StaticResource resource_name}`. For example, the following XAML code makes use of the above resources to set the `TextBlock` color and content:

```

<TextBlock x:Name="text_block"
  Foreground="{StaticResource olive_color}"
  Text="{StaticResource ironruby_text}" />

```

Accessing the resources via code is an easy matter, too, but with one trick:

```
win.find_name("text_block").text = win.find_resource("ironruby_text").to_clr_string)
```

The `find_resource` method must retrieve `System::String` values. The IronRuby string consists of the `MutableString` class, and you end up not finding the requested resource if you use it directly. Make sure to use the `to_clr_string` method to convert IronRuby's string to the expected `System::String`.

## Styles

These static resources mostly consist of style elements. This is why WPF provides a way to gather style settings into a single resource. This makes it much easier to create and use systemwide styles.

A style declaration starts with a root element named `Style` (surprise) that is given the known `x:Key` for referencing. The child nodes of a `Style` element are `Setter` elements that have two attributes: `Property` and `Value`. The `Property` attribute defines the name of the property to set, and the `Value` defines the value to set.

For example, the following XAML code creates a style that sets the foreground color and text of the control that uses it:

```

<Window.Resources>
  <Style x:Key="the_ironruby_style">
    <Setter Property="TextBlock.Foreground" Value="Red" />
    <Setter Property="TextBlock.Text" Value="IronRuby!!!" />
  </Style>
</Window.Resources>

```

Using this style is just as easy:

```
<TextBlock Style="{StaticResource the_ironruby_style}"/>
```

As you can see, it might be irritating to write the target type on every property name (for example, `TextBlock.Foreground` and `TextBlock.Text`). The `TargetType` attribute is exactly what we need to skip it:

```
<Style x:Key="the_ironruby_style" TargetType="TextBlock">
  <Setter Property="Foreground" Value="Red" />
  <Setter Property="Text" Value="IronRuby!!!" />
</Style>
```

## Binding to Dynamic Data

Binding to dynamic data is one of the tasks you perform most often when programming user applications. This task is quite straightforward in WPF.

The first task is to set the data to the element. The binding is done to specific properties in the given data object.

For example, the next class represents a single person:

```
class Person
  attr_accessor :first_name, :last_name

  def initialize(first_name, last_name)
    self.first_name = first_name
    self.last_name = last_name
  end
end
```

### IRONRUBY CLASSES FOR DATA BINDING

You should be aware of a few issues when constructing classes that WPF should be bound to.

Binding is done to specific properties in the data object. IronRuby does not have an equivalent to CLR properties. Even attributes are only metaprogramming methods that create getter and setter methods.

The WPF framework has a workaround for it. It finds the needed properties within the IronRuby object if it has a getter and setter method. In every other case, WPF won't find the data property.

Another issue is the lack of full support of IronRuby types. If you encounter a problem with the data binding, try using CLR types rather than IronRuby ones.

---

To bind a WPF element to an instance of the `Person` class, the element's `data_context` property should be set:

```
shay = Person.new("Shay", "Friedman")
# Bind the entire window
win.data_context = shay
```

On the XAML side, similar to the way of binding to static resources, we use the special binding value format. This time, between curly brackets we use the `Binding` keyword and the property name afterward:

```
<TextBlock>
  Full name:
  <TextBlock Text="{Binding first_name}"/>
  <TextBlock Text="{Binding last_name}"/>
</TextBlock>
```

### Data Templates

In most scenarios, you bind more than just one value. Most of the time, we have to present a list of records taken from some kind of a data source.

WPF provides an integrated way to bind to lists of data called *data templates*. The idea is that instead of defining the way one item looks, we provide a template for all of them.

The binding in the template itself is done like you saw with the `Binding` keyword. This is because every presentation item has the right data item as its data context.

Let's look at a simple `ListView` data template:

```
<ListView x:Name="listbox" ItemsSource="{Binding}">
  <ListView.ItemTemplate>
    <DataTemplate>
      <TextBlock>
        <TextBlock Text="{Binding first_name}"/>
        <TextBlock Text="{Binding last_name}"/>
      </TextBlock>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

Notice that to “tell” the `ListView` that it is going to be data bound, its `ItemsSource` attribute contains a `{Binding}` directive. This is important because the binding will not happen otherwise.

When we have the UI ready, let's fill it up with data:

```
shay = Person.new("Shay", "Friedman")
john = Person.new("John", "Doe")
```

```
melissa = Person.new("Melissa","Smith")
win.find_name("listbox").data_context = [shay, john, melissa]
```

Eventually, with the addition of a window element that wraps the `ListView` and upon execution of the XAML code via code, the window shown in Figure 13.13 results.

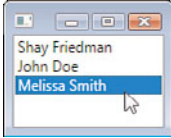


FIGURE 13.13 Using data templates.

## REPL

IronRuby, as mentioned in early chapters of this book, is a read-evaluate-print loop (REPL) language. This means that the language has integrated capabilities to evaluate code during runtime.

Adding REPL capabilities to an IronRuby-driven WPF application requires minor changes to the application. All we need actually is a place to write our code and a way to execute it (a button click or some other way).

For example, look at the following WPF application:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <StackPanel>
    <TextBlock x:Name="text_block" />
  </StackPanel>
</Window>
```

It is a simple application that contains one `TextBlock` named `text_block`. We now add REPL capabilities to this application.

First, let's add a text box and a button so that we have a place to insert Ruby commands:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <StackPanel>
    <TextBlock x:Name="text_block" />
    <TextBox x:Name="ruby_command" />
    <Button x:Name="btn">Execute</Button>
  </StackPanel>
</Window>
```

The second and last task is to handle the button's `click` event via code. There we evaluate and execute the code:

```
app = System::Windows::Application.new
win = System::Windows::Markup::XamlReader.parse xaml
win.loaded {
  win.find_name("btn").click {
    code = win.find_name("ruby_command").text.to_s
    eval code
  }
}
app.run win
```

`eval` is the method that executes textual Ruby commands. Notice that it receives a Ruby string and the text box contains a `System::String` value. This is why the `to_s` method should be called before passing the value to the `eval` method.

Now we can pass commands and execute them during runtime. For example, we can change the window height by filling `win.height = 500` in the REPL box and clicking the button to execute it.

Adding REPL to a WPF application is easy. I recommend that you take advantage of this capability; it can be a lifesaver. You can access elements, read values, and see how design changes take place in a live and instant way.

## Summary

In this chapter, you learned the fundamentals of WPF via IronRuby. WPF is a powerful framework that brings fresh air to Windows applications development.

You have learned in this chapter about window types, layout options, graphics capabilities, and data binding. The WPF framework features even more (templates to fully control UI elements, 3D graphics, themes, printing capabilities, and more, and then some more).

If you liked what saw in this chapter, I urge you to read more about it. A good place to start is the Microsoft website for Windows client programming at <http://windowsclient.net>.

# CHAPTER 14

## Ruby on Rails

Ruby has been around since 1995. Yet it was not widely known until 2006, the year when Ruby on Rails became popular. Ruby on Rails, also known as Rails or RoR, is an innovative web framework. It has brought new concepts to web development and influenced the entire web development industry. Clones started to show up in various languages like PHP, Java, Python, and .NET.

The framework started as an internal product of 37signals, led by David Heinemeier Hansson. In 2004, they decided to make the project open source, and the rest is history.

Nowadays, many websites are developed using Ruby on Rails. The biggest and most popular one is Twitter, which receives millions of visitors every day.

This chapter introduces you the main concepts of Ruby on Rails. In this chapter, you build a small, full-featured web page by exploiting Rails features.

### Preparing Your Environment

Ruby on Rails is a RubyGem. Therefore, the gem should be installed before you can start developing Ruby on Rails applications. In addition, RoR applications use databases to store the data of the application, so you need a database server as well. Follow the next steps to prepare your computer for RoR development:

#### IN THIS CHAPTER

- ▶ Preparing Your Environment
- ▶ Hello, IronRuby on Rails
- ▶ The Basic Concepts
- ▶ Main Components
- ▶ Know Your Environment
- ▶ Creating a Page
- ▶ Creating a Database-Driven Page

1. Install the Ruby on Rails gem. This is done with the `igem` tool, which you can find in the IronRuby installation folder. To install RoR, run the following command on the command prompt:

```
> igem install rails
Successfully installed activesupport-2.3.3
Successfully installed activerecord-2.3.3
Successfully installed rack-1.0.0
Successfully installed actionpack-2.3.3
Successfully installed actionmailer-2.3.3
Successfully installed activeresource-2.3.3
Successfully installed rails-2.3.3
7 gems installed
...
```

2. Make sure that the IronRuby installation folder is on the `PATH` environment variable. On the command prompt, write `set PATH` to see all paths that are listed there. To add the IronRuby folder to the `PATH` variable, use the following command (assuming that IronRuby is installed at `C:\IronRuby`):

```
set PATH = %PATH%;C:\IronRuby
```

3. Set up the database adapter. RoR has support for the major databases. If it doesn't support one, there is a good chance that a connection adapter has already been written for it. In this chapter, we use the SQL Server adapter that was written by the IronRuby team in C#.

To use it, you must first install SQL Server or the free SQL Server Express. You can download the latest version from <http://www.microsoft.com/express/sql/default.aspx>.

When SQL Server is installed and running, download the MS SQL connection adapter from <http://github.com/jschementi/activerecord-mssql-adapter>. The downloadable zip file contains a file named `mssql_adapter.rb`. Copy this file to the `%IronRuby Gems folder%\ 1.8\gems\activerecord-2.3.3\lib\active_record\connection_adapters` folder.

Your environment is now ready to run Ruby on Rails applications. These steps should be done only once per machine so once they are done, there is no need to redo them for every new application.

## Hello, IronRuby on Rails

In this section, we create our first Ruby on Rails web application that is driven by IronRuby. It is a basic application for every IronRuby on Rails application. Therefore, you can use this section to create the initial structure of every IronRoR application.

## Creating the Initial Project Files

The first task is to create the project files. Ruby on Rails does that for us, and with a single command, we can have an almost entirely working application.

To do that, open the command prompt in the location where you want the project file:

```
> cd MyProjects
```

```
MyProjects>
```

Now we need to create the actual project files. We use the irails tool for that matter. It takes the project name as an argument. Make sure the name doesn't contain spaces:

```
MyProjects> irails IronRubyRocks
      create
      create  app/controllers
      create  app/helpers
      create  app/models
      create  app/views/layouts
      create  config/environments
      ...
      create  public/javascripts/application.js
      create  doc/README_FOR_APP
      create  log/server.log
      create  log/production.log
      create  log/development.log
      create  log/test.log
```

The irails tool creates a set of folders and files that actually sum up to a small web application.

## Directory Structure

The numerous folders created for every new RoR application might be confusing at the beginning. Table 14.1 tries to clear it up and provides information about the folder structure and the role of each folder.

TABLE 14.1 Directory Structure Description

| Folder Name     | Description  |
|-----------------|--|
| App             | The main code of the application lies within the subdirectories of this folder. As a directory of itself, it doesn't have a specific role. |
| app/controllers | Holds the controller classes, which are responsible for handling web requests.   |



TABLE 14.1 Directory Structure Description

| Folder Name        | Description   |
|--------------------|---|
| app/helpers        | Holds helper classes that are available for controller, model, and view classes.  |
| app/models         | Contains the model classes, which are responsible for the application business logic.   |
| app/view           | Contains the HTML templates for viewing the application data.   |
| app/view/layouts   | View layouts that repeat on various different views can be put here for reuse.  |
| Config             | Configuration files for the application-like database configuration files.  |
| Db                 | Holds database access classes.  |
| Doc                | Contains the application documentation after <code>irake doc:app</code> is executed.  |
| Lib                | Contains code that doesn't fit one of app subdirectories.   |
| Log                | Error logs will be created here.  |
| Public             | Contains static pages and resources like 404 error page.  |
| public/images      | Contains the application static images.   |
| public/javascripts | Holds the application static JavaScript code.   |
| public/stylesheets | Contains the application static CSS code.   |
| Script             | Contains rails scripts. For example, <code>server</code> runs the web server.   |
| Test               | Contains the test code for the application. There are subdirectories for fixture, functional, integration, performance, and unit tests. |
| Tmp                | Temp files directory used by the Rails framework.   |
| Vendor             | Holds third-party libraries and plug-ins.   |

## Database Configuration

Ruby on Rails web applications are data-driven applications. This means that they are connected to a database. The RoR framework makes it easy to work with the database, including creating the needed tables.

Before we can start and run our first Rails application, we need to create a database. We already have SQL Server or SQL Server Express installed, so we can go on and create the database.

Follow the next steps to create the SQL database:

1. Go to application folder\config\database.yml and open it for editing. You see YAML code in there that contains database configuration. There is different configuration for the development, testing, and production environments. The default database is SQLite, but we want to use SQL Server.

To change it, replace the development part so that it looks like the following:

```
development:
  adapter: mssql
  host: localhost\SQLEXPRESS
  database: IronRubyRocks
  integrated_security: true
```

Save the file and close it.

2. Now we create the database:

- Open the SQL Server Management Studio application (Start menu > All Programs > Microsoft SQL Server > SQL Server Management Studio).
- Log in to the database with your credentials (Windows or SQL Server). If you work with SQL Express, the connection dialog looks similar to the one shown in Figure 14.1.



FIGURE 14.1 SQL Server login window.

- In Object Explorer on the left, right-click Databases and choose New Database.
- Write the name of the database. It should be the same name you have set in the database.yml file. Click OK to finish.

Note that apart from creating the database, there is no need to take any more actions directly on the database. Ruby on Rails provides tools for adding and designing database tables, which are discussed in further detail in the following sections.

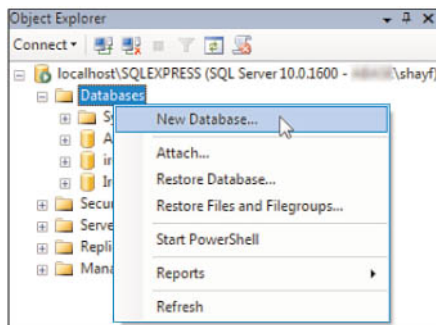


FIGURE 14.2 Creating a new database from “Object Explorer”.

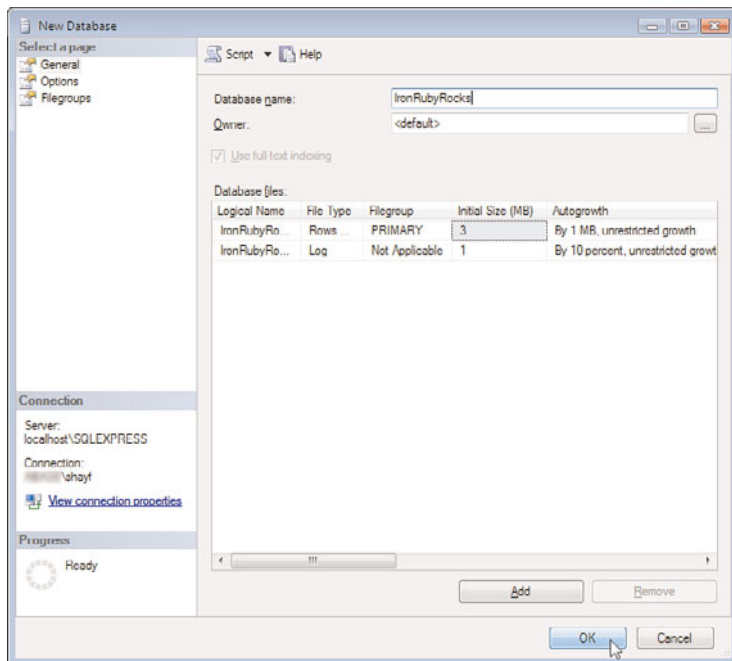


FIGURE 14.3 Naming the new SQL database.

## CREATING A DATABASE FROM THE COMMAND LINE

For some databases, like SQLite, the creation of a new database can be done with RoR command-line tools.

All you have to do is execute the irake tool with db:create as its argument, as follows:

```
irake db:create
```

It's important to note that this does not work for SQL Server and some other databases. If it doesn't work for you, just create the database yourself and continue.

---

## Running the Server

To run the Ruby on Rails application, we use the command line to start the server and then log in to the web page.

To start the server, open the command prompt and navigate to the web application folder. The command that starts the server is `ir script/server`:

```
> cd MyProjects\IronRubyRocks
MyProjects\IronRubyRocks> ir script/server
=> Booting WEBrick
=> Rails 2.3.3 application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2009-07-31 11:38:04] INFO  WEBrick 1.3.1
[2009-07-31 11:38:04] INFO  ruby 1.8.6 (2008-05-28) [i386-mswin32]
[2009-07-31 11:38:04] INFO  WEBrick::HTTPServer#start: pid=6368 port=3000
```

The server is now running and ready to accept requests. Therefore, we can visit our first Ruby on Rails web application.

Open your browser and navigate to <http://localhost:3000>. The result will be similar to the one in Figure 14.4.

Now click the About Your Application's Environment link. When you click it, an AJAX call to the server is sent, and some information is delivered back to you, as shown in Figure 14.5.

Your first Ruby on Rails application is now complete. Note that this application is the basic one for every RoR application; hence you can start every RoR application by building this basic application and then enhancing it.

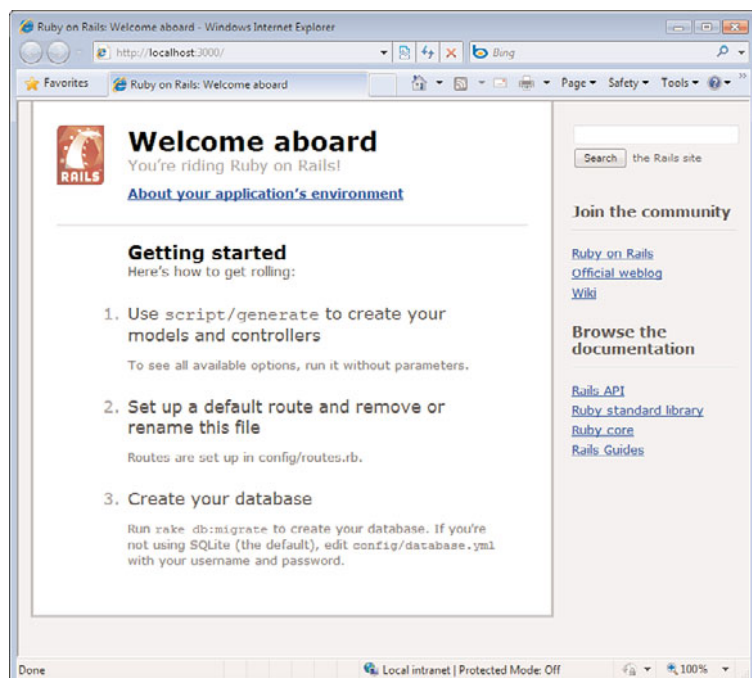


FIGURE 14.4 The default page of every Ruby on Rails application.

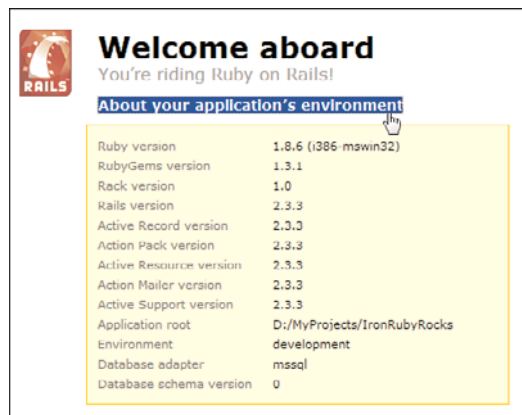


FIGURE 14.5 The “application environment” part of the default page.

## The Basic Concepts

Ruby on Rails has a few main concepts and guidelines you should be aware of before using it. These concepts include MVC, REST, CoC, and DRY.

### MVC

MVC is a design pattern. It stands for model-view-controller. Its aim is to completely separate the business logic from the presentation. Ruby on Rails applications are built with this design pattern. (Remember the app folder subdirectories?)

The model is the main layer of this architecture. It contains the business logic and the expected responses to changes to its state. It means that all the core functionality of the application will be placed here (for example, database reading/writing, generation of entities, and so on). A single model can serve various controllers and views.

The view is the user interface. It manages the display of information and reacts to state changes from the model. In Rails, views are mostly HTML files with embedded Ruby code.

The controller is the bridge between the model and the view. It reacts to user actions and acts accordingly, informing the model or the view when needed. The controller is responsible for the application behavior, and it is expected that a single controller will deal with a single functionality feature. As a result, multiple controllers can serve a single view. In Rails, controllers process incoming requests from the web browser.

### REST

REST stands for Representational State Transfer. It is a set of rules that instruct how web standards should be used (URIs, for example).

The center of the REST architecture is resources. A resource might be an application entity, an image, a web page, or any other “thing” you might come up with.

REST principals important to Ruby on Rails include the following:

- ▶ **Every resource has an ID:** This is a good practice in any application, not just REST applications. When you have a unique identifier for a resource, you can refer it from any place it might be needed.

For example, the following URL shows a page for the user whose ID is 987 (assuming this page is implemented):

`http://localhost:3000/users/987`

- ▶ **Link state:** The ability to transfer the state of the resource between multiple system components.

The REST theory is bigger than what I have introduced to you. If you want to read more about it, check out REST on Wikipedia: [http://en.wikipedia.org/wiki/Representational\\_State\\_Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer) or Roy Fielding doctoral dissertation at <http://www.ics.uci.edu/~taylor/documents/2002-REST-TOIT.pdf>.

## CoC

CoC is a design paradigm. It stands for Convention over Configuration. For the Rails developer, it means that you need to treat only the unconventional aspects of the application. Rails assumes what you want to do and how you're going to do so instead of letting you set multiple configuration files to achieve just that.

The standard convention of RoR is that the model is named after the table. For example, if the table is named `user`, the model will be named `User`, too. The controller will be named `users_controller`, and the views will contain a folder named `users` with template pages inside such as `new`, `edit`, and `show`. The convention is presented in Table 14.2.

TABLE 14.2 Default Ruby on Rails Convention

| Database Table Name | User  |
|---------------------|---|
| Model               | <code>app/models/user.rb</code>   |
| Controller          | <code>app/controllers/users_controllers.rb</code>   |
| Views               | <code>app/views/users/show.html.erb</code><br><code>app/views/users/edit.html.rb</code><br><code>app/views/users/new.html.rb</code> |

## DRY

The last acronym for this section is DRY: Don't Repeat Yourself. The idea is simple. After you have written code, you do not need to repeat it. RoR helps by reducing duplicated code blocks, which makes your application code more readable and less error prone.

## Main Components

Now that you know what the main concepts of Ruby on Rails are, it's time to take a look at the building blocks of every Ruby on Rails application.

### The Model: ActiveRecord

`ActiveRecord` is the base class for all models in an RoR application. It provides the business logic of the application and is supposed to be the main and biggest part of the application code.

ActiveRecord is an ORM (Object Rational Mapping) layer. It provides indirect and database-independent access to the data storage. The principles are simple:

- ▶ Database tables are mapped into classes.
- ▶ Table rows are mapped into objects.
- ▶ Table columns are mapped to attributes.

ActiveRecord provides CRUD functionality (create, read, update, and delete objects on the database), advanced find capabilities, the capability to relate models to one another, and more.

## The View: ActionView

ActionView manages the view layer of the application. It can create HTML and XML output, use layouts and partials, execute helper class methods, and generate AJAX calls.

## The Controller: ActionController

The base class for the RoR controllers is ActionController. Rail's controllers, unlike its models, are aware of the fact that the client is actually a web browser and they are capable of interpreting web requests. Incoming requests are processed by them, and the related action is invoked with the extracted parameters.

ActionController provides session and cache management tools, rendering view template capabilities and redirecting capabilities.

## Routes

Routing is the first meeting point of the Ruby on Rails application. Just like a real router that receives network requests and routes them to the right port, routing in RoR retrieves incoming web requests and sends them to the right controller to get going.

The action takes place in the config\routes.rb file. If you open it, you see lots of lines that use the map object.

On the last lines of the file, you can find the default routes, which map the request to its obvious controller (request for “users” will be mapped to users\_controller, for instance).

The routes.rb file exists not for the obvious mapping of course. It is there for customization reasons. For example, if I would like to change the convention (not recommended!) and call the people controller when a request to view a user page is sent, I'd add the following line:

```
map.connect 'users/:id', :controller => 'people', :action => 'view'
```

The result of this line is that when a visitor navigates to <http://localhost:3000/users/15>, the view method in app/controllers/people\_controller.rb file will be invoked.



Routing is much wider than this brief description. If you are interested in reading more about it, the Rails webpage provides a great in-depth guide: <http://guides.rubyonrails.org/routing.html>.

## Know Your Environment

Before you start creating real applications, you need to know a bit more about how to interact with the RoR environment.

The foremost point you need to know is that most interaction is done via the command line. You can execute several different commands: creating database tables, creating views and controllers, and more. The following subsections introduce you to the available commands.

### script/server

The `script/server` command is the one that runs the Ruby on Rails web server. It is needed when your web server is not one that is always up (IIS, for instance).

The command format is `ir script/server [options]`.

The options part is optional. You can call `ir script/server` without any options and the default options will be used. Table 14.2 describes the commonly used options of the `script/server` command.

TABLE 14.2 `script/server` Common Command Options

| Option                    | Description  |
|---------------------------|--|
| -p<br>or<br>--port        | The port to use for the server. Default is 3000.<br>Sample:<br>> <code>ir script/server --port=5678</code>   |
| -u<br>or<br>--debugger    | Enables Ruby debugging.  |
| -e<br>or<br>--environment | Runs the server in the given environment configuration: development, test or production. Default is development.<br>Sample:<br>> <code>ir script/server -e=production</code> |

Note that the preceding options can be mixed together. For example, to run a production server on port 5678, the command is as follows:

```
> ir script/server --environment=production -p=5678
```

## script/generate

Some of the most used commands are those that generate stuff. This is what the `script/generate` command does. It can generate almost anything in the application and make it available to you in a matter of seconds.

The command format is `ir script/generate [option] [arguments]`.

Table 14.3 describes the commonly used options of the `script/generate` command.

TABLE 14.3 `script/generate` Common Command Options

| Option     | Description  |
|------------|--|
| controller | <p>Create a new controller in <code>app/controllers</code>, view templates in <code>app/views/controller_name</code>, a helper class in <code>app/helpers</code>, a functional test suite in <code>test/functional</code>, and a helper test suite in <code>test/unit/helpers</code>.<br/> Format (operation list is optional):<br/> <code>ir script/generate controller [controller name] [operations]</code><br/> Sample:<br/> <pre>&gt; ir script/generate controller book sell buy read toc       exists  app/controllers/       exists  app/helpers/       create  app/views/book       exists  test/functional/       create  test/unit/helpers/       create  app/controllers/book_controller.rb       create  test/functional/book_controller_test.rb       create  app/helpers/book_helper.rb       create  test/unit/helpers/book_helper_test.rb       create  app/views/book/sell.html.erb       create  app/views/book/buy.html.erb       create  app/views/book/read.html.erb       create  app/views/book/toc.html.erb</pre></p> |
| model      | <p>Creates a new model in <code>app/models</code>, a unit test in <code>test/unit</code>, a text fixture in <code>test/fixtures</code>, and a migration in <code>db/migrate</code>.<br/> Format (attribute list is optional and consists of space separated items in format of <code>column_name:sql_type</code>):<br/> <code>ir script/generate model [model name] [attribute list]</code><br/> Sample:<br/> <pre>&gt; ir script/generate model book name:string price:int       exists  app/models/       exists  test/unit/       exists  test/fixtures/       create  app/models/book.rb       create  test/unit/book_test.rb       create  test/fixtures/books.yml       create  db/migrate       create  db/migrate/20090801145631_create_books.rb</pre></p>   |

TABLE 14.3 script/generate Common Command Options

| Option   | Description  |
|----------|--|
| helper   | <p>Generates a helper class. It generates the class under <code>app/helpers</code> and a test suite in <code>test/unit/helpers</code>.</p> <p>Format:</p> <pre>ir script/generate helper [helper name]</pre> <p>Sample:</p> <pre>&gt; ir script/generate helper MyHelper       exists  app/helpers/       exists  test/unit/helpers/       create  app/helpers/my_helper_helper.rb       create  test/unit/helpers/my_helper_helper_test.rb</pre>  |
| scaffold | <p>Generates an application resource. This operation creates and prepares every aspect of the resource: a model, a db migration script, a controller, views, a stylesheet, a helper, and a full test suite.</p> <p>Format (attribute list is optional and consists of space separated items in format of <code>column_name:sql_type</code>):</p> <pre>ir script/generate scaffold [model name] [attribute list]</pre> <p>Sample:</p> <pre>&gt; ir script/generate scaffold user full_name:string bio:text age: integer       exists  app/models/       exists  app/controllers/       exists  app/helpers/       create  app/views/users       exists  app/views/layouts/       exists  test/functional/       exists  test/unit/       exists  test/unit/helpers/       exists  public/stylesheets/       create  app/views/users/index.html.erb       create  app/views/users/show.html.erb       create  app/views/users/new.html.erb       create  app/views/users/edit.html.erb       create  app/views/layouts/users.html.erb       create  public/stylesheets/scaffold.css       create  app/controllers/users_controller.rb       create  test/functional/users_controller_test.rb</pre> |

TABLE 14.3 script/generate Common Command Options

| Option   | Description  |
|----------|--|
| scaffold | <pre> create app/helpers/users_helper.rb create test/unit/helpers/users_helper_test.rb route map.resources :users dependency model exists app/models/ exists test/unit/ exists test/fixtures/ create app/models/user.rb create test/unit/user_test.rb create test/fixtures/users.yml exists db/migrate create db/migrate/20090801152012_create_users.rb </pre> |

The available column types are binary, boolean, date, datetime, decimal, float, integer, primary\_key, string, text, time, and timestamp.

If you want to find out about more available script/generate commands, run `ir script/generate` to get the help content. For specific command help, run `ir script/generate [command_name] --help`.

## ROLLING BACK

If you use the script/generate command to create resources but then decide you shouldn't have done so and want to remove the resource, RoR makes it easy for you: Just use the script/destroy command.

The script/destroy command receives the same attributes as script/generate, but without the extended attributes.

For example, if you use the following scaffold command

```
ir script/generate scaffold user full_name:string bio:text age:integer
```

You can roll back the operation (actual meaning = delete the files) with the following command:

```
ir script/destroy scaffold user
```

## db:migrate

When you use a scaffold command, for example, it generates a db migration script. This script contains the needed operations to create a database table, add/remove columns, and to perform other database operations.

The script isn't run instantly, and you need to run the `db:migrate` command to run it. The command is using the `irake` tool:

```
Application directory> irake db:migrate
```

For example, the following command prompt text demonstrates the `db:migrate` command for the `scaffold` command we executed in the previous section:

```
D:\MyProjects\IronRubyRocks>irake db:migrate
(in D:/MyProjects/IronRubyRocks)
== CreateUsers: migrating =====
-- create_table(:users)
   -> 0.3400s
== CreateUsers: migrated (0.4050s) =====
```

## Creating a Page

The previous section covered common commands for the RoR developer. With those commands in your arsenal, you can start building web pages using the Ruby on Rails framework.

The first page we create here is a simple page with no need for database access. It shows us the current time in several locations around the world.

## Generating the Page Controller and View

Creating a controller and a view involves executing a `script/generate controller` command:

```
MyProjects\IronRubyRocks> ir script/generate controller clocks
exists app/controllers/
exists app/helpers/
create app/views/clocks
exists test/functional/
exists test/unit/helpers/
create app/controllers/clocks_controller.rb
create test/functional/clocks_controller_test.rb
create app/helpers/clocks_helper.rb
create test/unit/helpers/clocks_helper_test.rb
```

As you can see, the command has created the controller (`clocks_controller.rb`) and the views folder.

Let's take a closer look. If we open the controller file, we find an empty class:

```
class ClocksController < ApplicationController
end
```

This is the result of the lack of a view. We now add the available actions ourselves and implement them.

The first action I want to add is the `index` action. This default action takes place when no specific action is defined. This means that `http://localhost:3000/clocks` work, but `http://localhost:3000/clocks/show`, for example, ends up with an exception about an unknown action.

On the `index` action, I want to generate the data for the view. The data should contain a list of places and their current time. To do that, I calculate the current time in the UTC zone and then convert the time to the local time zone by adding or subtracting seconds from it:

```
def index
  utc_time = Time.now.getutc
  @clocks = []
  @clocks << ["USA - New York", utc_time - 60*60*5]
  @clocks << ["England - London", utc_time]
  @clocks << ["Israel - Jerusalem", utc_time + 60*60*2]
  @clocks << ["India - Mumbai", utc_time + 60*60*5.5]
end
```

As you can see, I store the data in an instance variable `@clocks`. I do this so that the view can access the controller's instance variables.

And speaking of the view, we have our controller doing its thing, so let's move on to the view. The view files are HTML files with Ruby code inside: *ERB* files. The `.erb` extension indicates eRuby, a templating system that allows embedding Ruby into text documents. eRuby isn't used only for HTML templating or Ruby on Rails; you can find it on other frameworks and in different usage scenarios, as well.

So, we need to add a matching view to the controller. To conform to the convention, the view should follow the name of the controller action. Hence, the file will be named `index.html.erb`. We create this file in the matching directory that has been created for us at the time we generated the controller: `app/views/clocks`.

Just by creating the file, we complete the circle and can immediately try to navigate to the page, `http://localhost:3000/clocks`.

This now gives us a blank page, of course, because we haven't put any content inside the `index.html.erb` file. Let's take care of it now.

We want to display the clocks and city names near each other. The ERB content can do that:

```
<html>
<head>
  <title>Clocks</title>
</head>
<body>
  <%= @clocks.each do |city, time| %>
    <b><%=h city %></b>: <%=h time %>
    <br/>
  <% end %>
</body>
</html>
```

Note the Ruby code inside of this HTML code. The Ruby code goes between the `<% %>` symbols. The `<%= %>` outputs the result of the code that is written inside.

### MAKING VALUES HTML SAFE

When inserting text into HTML documents, be careful not to insert characters that might damage the HTML content (`<` or `>` characters, for example).

Ruby on Rails provides a method named `h` that make the text HTML safe by converting special characters into its HTML entities (for example, `<` is converted to `&lt;`).

Just like in the `index.html.erb` sample, make sure to use the `h` method when using the `<%= %>` symbols. For example, the following code

```
<%=h " <hello>"%>
```

is translated to

```
&lt;hello&gt;.
```

The preceding code loops over the `clocks` array that is built in the controller and prints every city in its own row. The output looks like Figure 14.6.

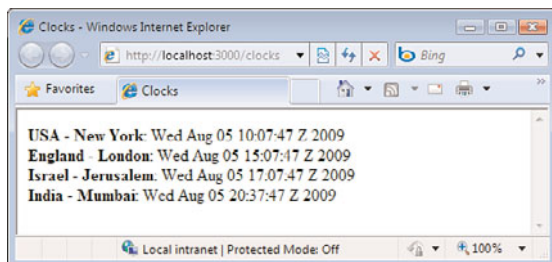


FIGURE 14.6 Output of clocks page showing the current local time in various places.

That's the page we wanted. We have the cities and the local time in each presented to us. The only issue here that bothers me is the unfriendly date format. I'd like to change that. The straightforward way is to change the view code and format the date to a friendlier representation. However, we want to keep the view as "dumb" as we can. The formatting method fits a different location: the helper class.

## Helper Classes

Helper classes are classes that are meant to help in common view-related tasks. Each controller has a helper class available for the views only; the controller cannot access it. Using helper classes also helps you follow the Don't Repeat Yourself dictum.

The `application_helper.rb` file contains helper methods that can be used on any view across the application, whereas other helpers are available only for their matching view.

We add our helper method to the `clocks_helper.rb` file:

```
module CLOCKSHelper
  def get_formatted_time(time)
    time.strftime("%B %d %Y %H:%M")
  end
end
```

With that in place, we can go back to our view and update the code:

```
<% @clocks.each do |city, time| %>
  <b><%=h city %></b>: <%=h get_formatted_time(time) %>
  <br/>
<% end %>
```

Now the output looks much better, as you can see in Figure 14.7.

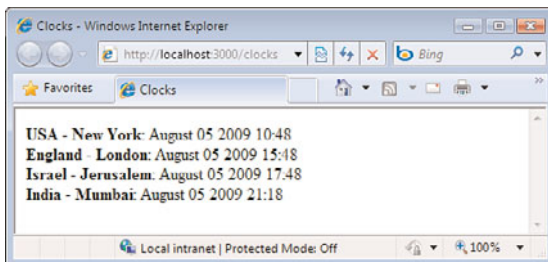


FIGURE 14.7 Presenting a user-friendly time string using helper classes.



**BUILT-IN HELPERS**

Ruby on Rails comes with multiple helpers that help in various view-related tasks. For example, `TagHelper` helps in building HTML output, `UrlHelper` helps in generating application URLs, and `FormHelper` helps in creating RoR HTML forms.

Look into it on the Ruby on Rails API site (look for classes that start with `ActionView::Helpers`): <http://api.rubyonrails.org>.

**Adding Stylesheets**

Our page is very functional now, but it needs some more color to it. Stylesheets are the answer to our colorful needs.

Unlike RoR objects so far, stylesheets are not page specific. This is logical because most of the times you would want a few CSS files that set the entire application theme. You still can, however, add CSS code to a specific page only; you have the HTML page itself, and you can decide what to do with it.

The place for the application stylesheets is `public/stylesheets`. Let's add a new file there, `blue_theme.css`, with the next content:

```
BODY {
  font-size: 14px;
  color: white;
  background-color: blue;
}

B {
  font-size: 16px;
  font-weight: bold;
  color: chartreuse;
}

a { color: aqua; }
img { border-width: 0px; }
```

This CSS code makes our page very colorful. To make the page use it, we edit the `index.html.erb` file and update the header file with a stylesheet reference:

```
<head>
  <title>Clocks</title>
  <%= stylesheet_link_tag "blue_theme" %>
</head>
```

## STYLESHEET\_LINK\_TAG

Remember the helpers we examined earlier in this chapter? `stylesheet_link_tag` is one of the built-in helpers available for every view in the application. The specific helper that provides this method is `AssetTagHelper`.

After this change is applied, the page becomes colorful, as presented in Figure 14.8.

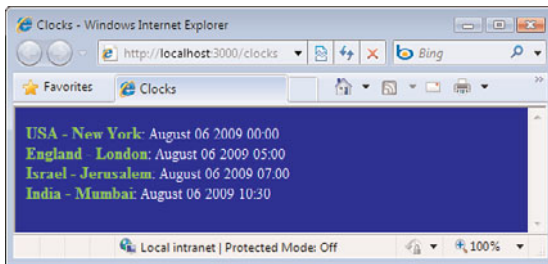


FIGURE 14.8 The page after CSS is applied.

## Adding Layouts

Layouts are HTML pages that contain layout components that appear on multiple pages. Layouts can save you a tremendous number of lines of code in big applications, all while letting you adhere to the DRY dictum.

A layout is just like any view. It's an ERB file, and RoR uses, by default, the layout with the name of the controller. If this is not found, it looks for a layout named `application.html.erb`, which is the default layout of the application. The only difference is that layout exists in `app/views/layouts`.

I'd like to create an applicationwide layout, so let's add `application.html.erb` file to the layouts folder with the following content:

```
<html>
<head>
  <title>IronRuby Rocks: <%= @title %></title>
  <%= stylesheet_link_tag "blue_theme" %>
</head>
<body>
  <h1 style="text-align:center">IronRuby Rocks Web Site</h1>
  <hr>
```

```

    <%= yield %>
  </body>
</html>

```

Let's see what each part of this layout contains.

### Head Part

The head part contains the stylesheet we used in the page. It is there because it is a cross-application style, so there is no reason to add it on each view separately.

The addition to the head part is the title part. I create the title by concatenating a static string "IronRuby Rocks:" and a view title, which will be provided by the controller.

To conform to that, the controller code will be extended:

```

def index
  @title = "Clocks"

  utc_time = Time.now.getutc
  @clocks = []
  ...
end

```

### Body Part

The body part contains a big header that appears on every view and indicates to the visitor what website she is currently viewing. (And believe me, visitors won't overlook it.)

The next part is the most important one: `<%= yield %>`. This is the location where the view itself appears. Every layout must have this to make the real content of the application available. Without it, the only content we see in our application is the layout (for every single view).

According to its new location within the layout page, we can update `clocks.index.html`. This is the view code after the update:

```

<% @clocks.each do |city, time| %>
  <b><%=h city %></b>: <%=h get_formatted_time(time) %>
  <br/>
<% end %>

```

Figure 14.9 shows what the view looks like now.

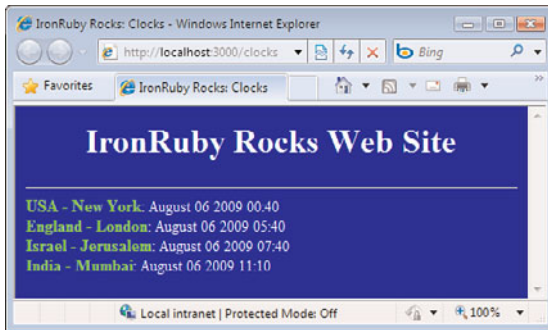


FIGURE 14.9 The page after the layout is applied.

## Adding Functionality

Now I want to add some functionality to this page.

The obvious feature for our page is a Refresh button. The first possibility that comes to mind is adding this code to the view:

```
<form><input type="submit" value="Refresh" /></form>
```

This actually works. It makes the form refresh. However, Rails provides a better mechanism for doing so.

One of the built-in helpers, `FormTagHelper`, can help us create HTML form tags by using Ruby commands. This is a better solution than the one at the beginning of this section because it also takes care of Rails-related issues (such as authentication) and enables more RoR capabilities (for example, enabling you to submit the form to a different action on the controller).

To achieve the Refresh button we're looking for, we add the following code snippet to the end of `index.html.erb` file:

```
<% form_tag do %>
  <%= submit_tag "Refresh!" %>
<% end %>
```

If we look at the source of the output HTML page, this is what we see:

```
<form action="/clocks" method="post">
  <div style="margin:0;padding:0;display:inline">
    <input name="authenticity_token" type="hidden"
value="8C0b/BStnkpTgjXc5IHRJmP7yrGTzGy4fJ4b5x71TZI=" />
  </div>
  <input name="commit" type="submit" value="Refresh!" />
</form>
```

As you can see, Ruby on Rails enriched the form data to fit its needs. For example, the form action goes to the current page. (We could specify a different action that would have been redirected to a different controller method.) Rails added the `authenticity_token` hidden field, too, which is a part of its security mechanism.

## Creating a Database-Driven Page

The clocks page we built in the previous section ran on its own, without the need of a database. These pages do exist on web applications, but the bigger slice of the web pages pie belongs to database-driven pages.

In this section, we take the clocks page to the next level. We allow users to define and use their favorite clocks.

### Generating the Page Resources

We begin with the enhanced clocks page by executing the `scaffold` command to create the set of page resources. I call the model `UserClock` and use `City Name` and `Difference from UTC Time` columns:

```
> ir script/generate scaffold UserClock city_name:string utc_difference:float
```

After this command is done, we need to update the database with the new migration scripts:

```
> irake db:migrate
```

We're done. The page resources are done.

If you navigate to `http://localhost:3000/user_clocks` (assuming that the server is running, of course), you see that you already have a full operating page, as shown in Figure 14.10.

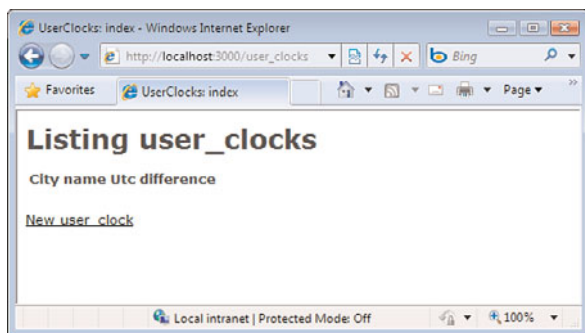
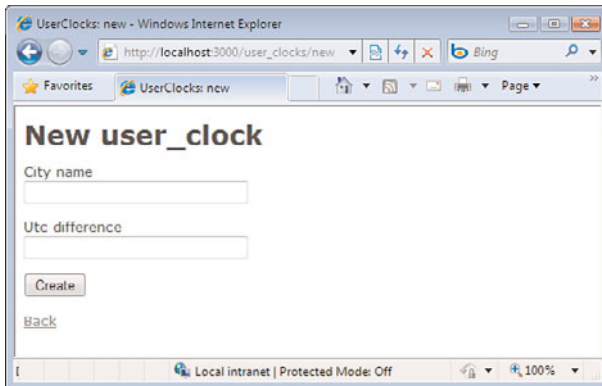


FIGURE 14.10 A default database-driven Ruby on Rails page.

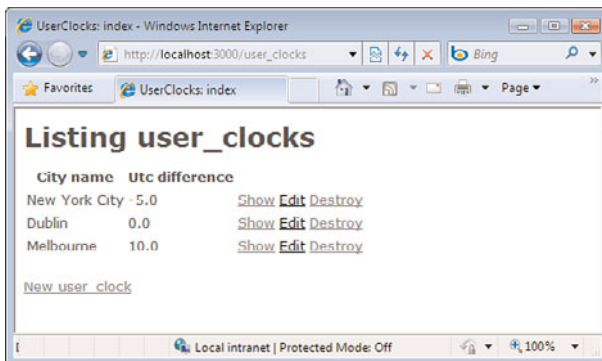
Clicking the New user\_clock link takes you to a form to insert a new user\_clock entity, as shown in Figure 14.11.



The screenshot shows a web browser window titled 'UserClocks: new - Windows Internet Explorer'. The address bar shows 'http://localhost:3000/user\_clocks/new'. The page has a title 'New user\_clock'. It contains two text input fields: 'City name' and 'Utc difference'. Below these fields is a 'Create' button and a 'Back' link. The status bar at the bottom indicates 'Local intranet | Protected Mode: Off' and '100%' zoom.

FIGURE 14.11 A default new record form.

Filling in the fields and clicking Create inserts a new record to the database and shows the new record on the index page. Figure 14.12 shows what the index page looks like after I have inserted a few clock definitions.



The screenshot shows a web browser window titled 'UserClocks: index - Windows Internet Explorer'. The address bar shows 'http://localhost:3000/user\_clocks'. The page has a title 'Listing user\_clocks'. It displays a table with two columns: 'City name' and 'Utc difference'. The table contains three rows of data: 'New York City' with '-5.0', 'Dublin' with '0.0', and 'Melbourne' with '10.0'. Each row has three links: 'Show', 'Edit', and 'Destroy'. Below the table is a link 'New user\_clock'. The status bar at the bottom indicates 'Local intranet | Protected Mode: Off' and '100%' zoom.

| City name     | Utc difference |   |
|---------------|----------------|---|
| New York City | -5.0           | <a href="#">Show</a> <a href="#">Edit</a> <a href="#">Destroy</a> |
| Dublin        | 0.0            | <a href="#">Show</a> <a href="#">Edit</a> <a href="#">Destroy</a> |
| Melbourne     | 10.0           | <a href="#">Show</a> <a href="#">Edit</a> <a href="#">Destroy</a> |

FIGURE 14.12 The default list page showing data from the database.

At this point, we have a fully functional page that allows listing, creating, editing, and deleting records. However, there are still a few tasks ahead to make this page really helpful to the user.

## Polishing the Index Page

The index page needs some polishing before it can become a real part of our application. The tasks ahead include the following:

- ▶ Word changes. `user_clock` should be replaced; `Destroy` should be renamed.
- ▶ Show the time, not the UTC difference.
- ▶ The list should have the layout of the list in the previous clocks view.
- ▶ The application layout should be used.
- ▶ A Refresh button should be added.

### Looking into the Generated View

Before we begin, we go through the generated index page and see how it is built.

To work on the index page, we need to open it for edit. The page is located at `app/views/user_clocks/index.html.erb`.

Looking into the source code is like revealing a magician's trick: You understand that there's no magic at all, only some quick fingers. Don't forget, however, that in software development, working quicker is important, and scaffolding helps us generate pages more quickly, much more quickly.

Anyway, back to our index page. The content of the view page is similar to the view we wrote by ourselves a few sections ago:

```
<h1>Listing user_clocks</h1>

<table>
  <tr>
    <th>City name</th>
    <th>Utc difference</th>
  </tr>

  <% @user_clocks.each do |user_clock| %>
    <tr>
      <td><%=h user_clock.city_name %></td>
      <td><%=h user_clock.utc_difference %></td>
      <td><%= link_to 'Show', user_clock %></td>
      <td><%= link_to 'Edit', edit_user_clock_path(user_clock) %></td>
      <td><%= link_to 'Destroy', user_clock, :confirm => 'Are you sure?', :method =>
:delete %></td>
    </tr>
  <% end %>
</table>
```

```
<br />
```

```
<%= link_to 'New user_clock', new_user_clock_path %>
```

First we loop over the data array and show the data by using the `<%= %>` symbols, exactly what we did earlier.

The improvement here is the links. Every clock contains three links: one to show the clock alone, one to edit its details, and one to delete it. At the lower part of the page is a link to create a new clock.

The `link_to` method, which is part of the `UrlHelper` class, generates a link to another RoR page. It is especially helpful with ActiveRecord objects (like `user_clock` in the preceding code block), where it can create links to specific object page or methods (like the link to the edit page or to the `delete` method).

### Word Changes

Now that you know what the index view actually looks like, changing the problematic words is a matter of seconds.

The header should be totally removed, and the label of the link at the end of the page should be changed to `Add a new clock`:

```
<%= link_to 'Add a new clock', new_user_clock_path %>
```

The `Destroy` link is also a bit too much. Let's soften it a bit and call it `Remove`:

```
<td><%= link_to 'Remove', user_clock, :confirm => 'Are you sure?', :method =>
  ↳:delete %></td>
```

We're done with the word changes. Our page is starting to look better, as you can see in Figure 14.13.

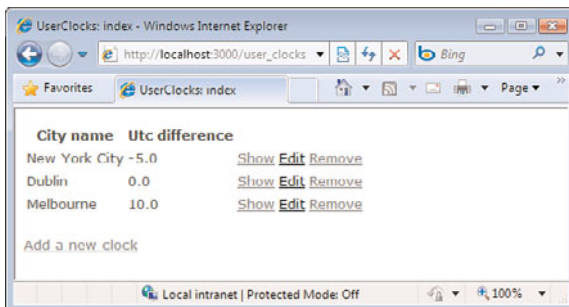


FIGURE 14.13 The index page after the word changes.



### Displaying the Time Rather Than the UTC Difference

To make the view more user friendly, we need to get the current time rather than the UTC difference value. The most suitable place for doing so is the view (because it is a view-only feature and not a data feature).

Let's open the `ViewHelper` module (located in `app/views/user_clocks_helper.rb`) and add a method that helps us transform the UTC difference value to a real-time object:

```
module UserClocksHelper
  @@utc_time = Time.now.getutc

  def convert_to_time(utc_difference)
    @@utc_time + (utc_difference.to_f * 60 * 60)
  end
end
```

Remember the `get_formatted_time` method we added to the `clocks_helper` class? We need it here, as well, to show the user a friendly time string. Because we need this method on two different views, it is a good idea to move it to the `application_helper.rb` file and place it in the `ApplicationHelper` module, where it will be available to all the views in the application:

```
module ApplicationHelper
  def get_formatted_time(time)
    time.strftime("%B %d %Y %H:%M")
  end
end
```

When we have these helper methods ready, we can move on to the view and update it. On the index view page, we update the `utc_difference` presentation code to use the helper methods we just created:

```
<td><%=h get_formatted_time(convert_to_time(user_clock.utc_difference)) %></td>
```

With these changes applied, the list displays the time rather than the UTC difference value, as shown in Figure 14.14:

### Applying the Previous List Layout

Scaffold's default table layout is nice, but it doesn't fit our needs for this view. I'd like to take the previous clocks view list layout and apply it here.

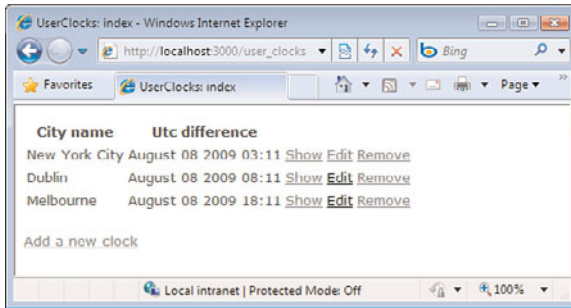


FIGURE 14.14 The list displays the time rather than the UTC difference.

Again, we open the `app/views/user_clocks/index.html.erb` file and update it. We're going to remove the table entirely and replace it with other HTML tags:

```
<% @user_clocks.each do |user_clock| %>
<b><%=h user_clock.city_name %></b>:
<%=h get_formatted_time(convert_to_time(user_clock.utc_difference)) %>

<%= link_to 'Show', user_clock %>
<%= link_to 'Edit', edit_user_clock_path(user_clock) %>
<%= link_to 'Remove', user_clock, :confirm => 'Are you sure?', :method => :delete %>
<br/>
<% end %>
```

This change makes the view a bit better, but we still have the links at the end of each line that seem not to belong. The way to solve this layout issue is by using images as links instead of plain text. We also remove the Show link because the Show action doesn't really have a meaning in our application. (It's good to remove the show method from the controller, too.)

Assuming that we have `edit.png` and `remove.png` files in the `public/images` folder, this is how the link part looks after the change has taken place:

```
<% link_to edit_user_clock_path(user_clock) do %>
  <%= image_tag "edit.png", :alt=> "Edit" %>
<% end %>

<% link_to user_clock, :confirm => 'Are you sure?', :method => :delete do %>
  <%= image_tag "remove.png", :alt=> "Remove" %>
<% end %>
```

With this change applied, Figure 14.15 shows what our page looks like now:

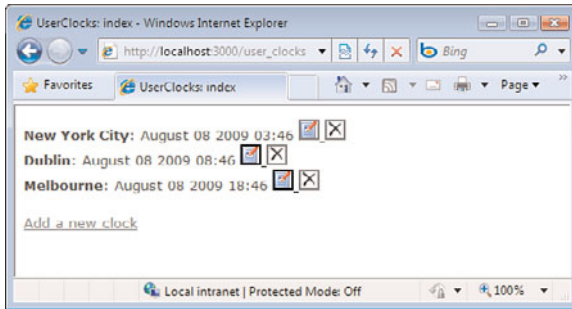


FIGURE 14.15 The previous list layout applied to the database-driven list page.

### Using the Application Layout

At this point, you probably wonder why our world-class application layout didn't take effect. This happens because the `scaffold` command generates a new layout page for the view. For the view to use the application layout, it must not have its own layout view.

Deleting the `app/views/layout/user_clocks.rb` file restores the application layout to our new page, as shown in Figure 14.16.

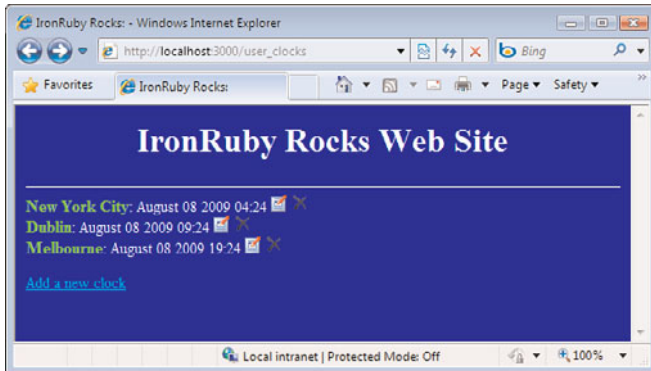


FIGURE 14.16 The list page after the application layout is applied.

### Adding the Refresh Button

For the Refresh button, we use a slightly different approach than what we have done on the previous clocks page.

We use the `button_to` method, which is a shorter version of what we did before. However, we need to set some options for our button to work:

```
<%= button_to "Refresh!", "/user_clocks", :method => :get %>
```

The first argument is the value of the button, the second is the path where the button should be submitted to, and the last is the method that the form uses.

Note that for the button to work, the method must be a `get` method. This is done to make the request similar to the regular one when users navigate to the page.

Figure 14.17 shows what the page looks like after the last addition.

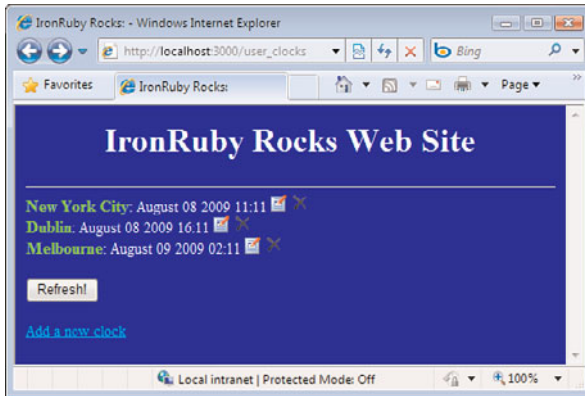


FIGURE 14.17 The page with the new Refresh button.

This is it. The list page is now complete. Users can add several clocks, modify them, and remove them at their will. Of course, they can also watch their clocks on the list page and see what the local time is in every city.

## Summary

In this chapter, you learned the basics of Ruby on Rails. You learned how to use the tools, create databases, generate web pages, and make use of the different sides of the MVC framework. You have also used stylesheets, layouts, and helper classes.

This chapter introduced you to the very basic basics of the RoR framework. Ruby on Rails is a wide and solid framework that enables you to do various different tasks in a matter of minutes. If you want to learn more about the Rails framework, a good place to start is its official site at <http://rubyonrails.org>.

*This page intentionally left blank*

# CHAPTER 15

## ASP.NET MVC

ASP.NET has been around for years without any significant change. With the rise of Ruby on Rails and other model-view-controller (MVC)-based web frameworks, Microsoft has joined the trend with its own MVC-based web framework. ASP.NET MVC is a framework built on top of the classic ASP.NET infrastructure. It is based on the model-view-controller design pattern and includes some new concepts such as routes, filters, and validations.

ASP.NET MVC is different from what you are used to with classic web forms. However, the new framework makes several tasks possible that have been hard (or even impossible) to implement before (for example, simple testing capabilities, AJAX, and improved code exit points).

This chapter covers the IronRubyMvc framework, which is the IronRuby implementation of the ASP.NET MVC framework. You learn how to use the MVC framework capabilities to create a simple web application.

And just a last word for you Ruby on Rails developers: You will find IronRubyMvc similar to the RoR framework. This is no surprise because they are both MVC-based web frameworks. However, each has its own advantages and disadvantages, so don't hesitate to learn IronRubyMvc; you might even like it!

### Preparing Your Environment

To use and run ASP.NET MVC on your computer, you need to install the ASP.NET MVC framework and the IronRubyMvc framework that gives ASP.NET MVC a native

#### IN THIS CHAPTER

- ▶ Preparing Your Environment
- ▶ Hello, ASP.NET MVC
- ▶ MVC
- ▶ Routes
- ▶ Filters
- ▶ Validations
- ▶ Classic ASP.NET Features
- ▶ A Word About Classic ASP.NET

Ruby feeling. Hence before we begin, you need to get your computer ready for IronRuby-driven ASP.NET MVC.

## Installing ASP.NET MVC

The first task is to download and install ASP.NET MVC from its official site: <http://www.asp.net/mvc/download>. This installation requires you to have Visual Studio 2008 SP1 or Visual Web Developer 2008 SP1 installed.

Visual Web Developer 2008 is free and can be downloaded from <http://www.microsoft.com/express/download>.

At the time of this writing, IronRubyMvc is built on top of ASP.NET MVC 1.0. If possible, use this version. Although it is supposed to be safe to run this on later frameworks, you might run into some unexpected behavior in case of major changes to the class library.

## Obtaining the IronRubyMvc DLL

For the next step, you need the IronRubyMvc DLL file. To get it, you must download its source code and compile it.

To compile the project, you must have Visual Studio or Visual C# Express.

Visual C# Express is free and can be downloaded from <http://www.microsoft.com/express/download>.

After you ensure you have a way to compile the code, follow these steps:

1. Navigate to <http://github.com/casualjim/ironrubymvc>.
2. Click the Download button and save the archive.
3. Extract the archive.
4. The extracted folder contains a folder named Dependencies. Go there and open `update.bat` for edit. Replace `c:\ironruby\bin` with the path to your IronRuby installation folder. After you're done, save the file and execute it.
5. Open `IronRubyMvc.sln` in Visual Studio and build the solution. Make sure you're building the `Release_Signed` configuration as shown in Figure 15.1.

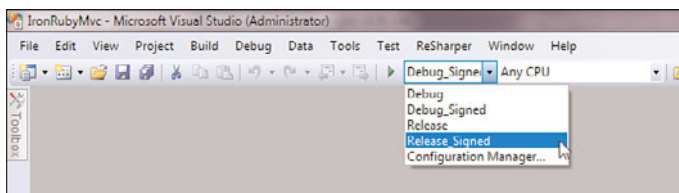


FIGURE 15.1 Choosing the `Release_Signed` build configuration in Visual Studio.

After the compilation is complete, you can close Visual Studio. The important files now lie within `ironrubymvc\bin\Release_Signed`. Remember this path; we use it in the next section.

## Adding IronRubyMvc Templates to Visual Studio

To make the work in Visual Studio smoother for the IronRuby developers, you can import and use a few templates.

The IronRubyMvc package you have just downloaded contains a folder named VS Templates. This directory is our starting point for the next steps:

1. Copy the content of the folder Project Template to My documents\Visual Studio 2008\Templates\ProjectTemplates\Visual C#.
2. Copy the content of the folder File Templates to My documents\Visual Studio 2008\Templates\ItemTemplates\Visual C#.

## Hello, ASP.NET MVC

ASP.NET MVC comes with a convenient Visual Studio integration. We can partially enjoy this convenience even though IronRuby doesn't play nicely with Visual Studio yet.

We can build the application structure inside Visual Studio and then continue with writing the actual code in the IronRuby IDE of our choice.

Throughout this chapter, we use IronRuby and ASP.NET MVC to build a simple page that enables us to manage a to-do list. Let's start by generating and running the initial project.

### Generating the Initial Project

Creating the initial project is done via Visual Studio by using the IronRubyMvc template. Follow the next steps to create the project

1. Open Visual Studio or Visual Web Developer and click File > New > Project.
2. In the New Project dialog, choose Visual C# on the left, and then choose at the bottom the template we prepared previously: IronRubyMvcTemplate. Type in the project name, **ToDoList**, choose a location, and click OK. The dialog should look similar to the one in Figure 15.2.
3. The project is now almost entirely ready. We now need to update it to the latest IronRuby and IronRubyMvc assemblies.

Open Solution Explorer and expand the References node. Delete the next assemblies from the project references: `IronRuby`, `IronRuby.Libraries`, and `System.Web.Mvc.IronRuby`, as shown in Figure 15.3.



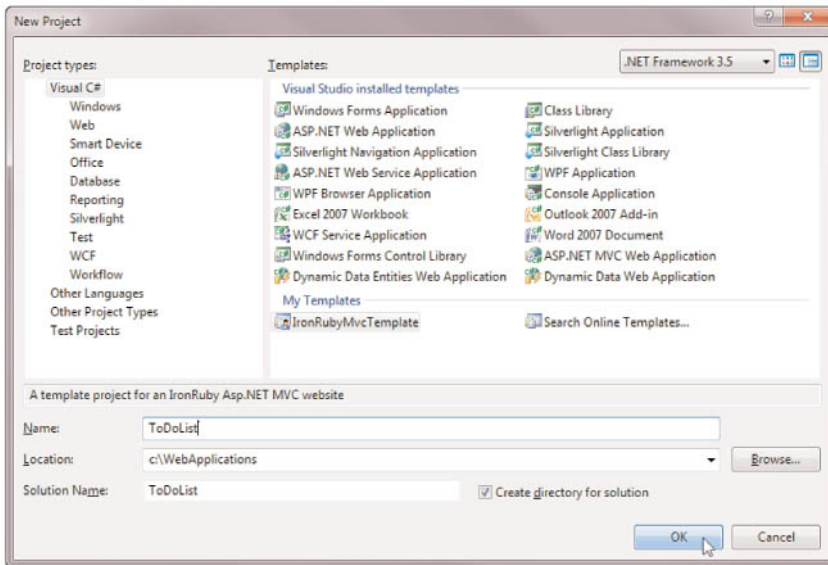


FIGURE 15.2 The New Project dialog with input.

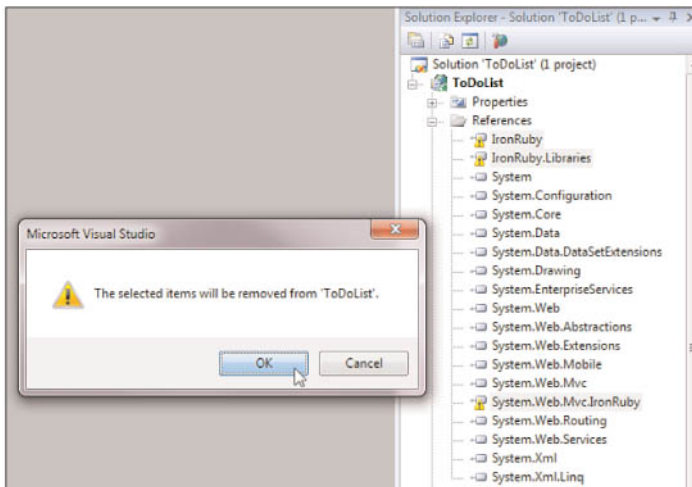


FIGURE 15.3 Removing the IronRuby assemblies.

4. Go to Project > Add Reference and browse to the IronRubyMvc\bin\Release\_Signed folder we prepared earlier. Choose the files IronRuby.dll, IronRubyLibraries.dll, and System.Web.Mvc.IronRuby.dll as shown in Figure 15.4 and click OK.

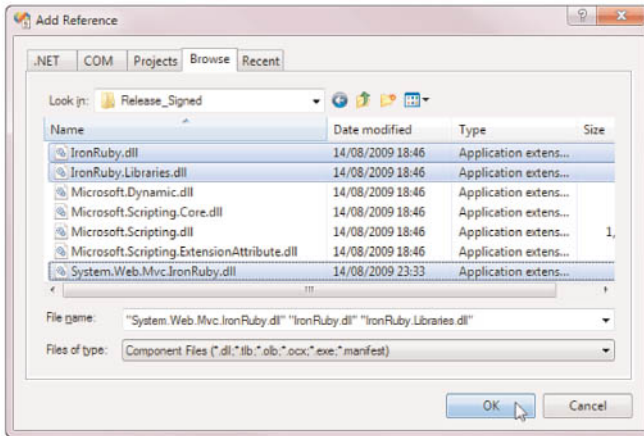


FIGURE 15.4 Adding references to the updated IronRuby assemblies.

5. We now need to fix an issue with the default controller. Open the file `Controllers/home_controller.rb` and replace the line. (It appears twice in the file.)

```
view nil, 'layout'
```

with this:

```
view '', 'layout'
```

6. Click F5 to run the web application using Cassini, Visual Studio's internal web server. The default ASP.NET page will be presented, as shown in Figure 15.5.

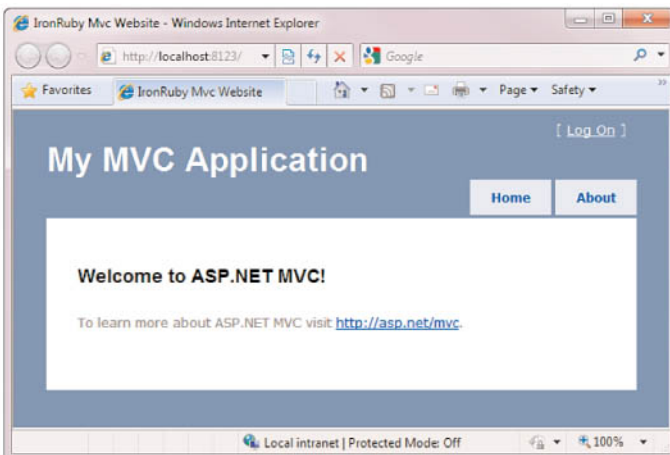


FIGURE 15.5 The default IronRuby-driven ASP.NET MVC page.

You have just run your first IronRuby-driven ASP.NET MVC application.

## MVC

MVC, which stands for model-view-controller, is a design pattern that makes it possible to separate an application into three layers, each with a different role.

### Models

Models in the MVC web application are the place where the business logic exists. They are plain IronRuby classes, with no special parent class or design rules. They are located in the Models folder.

Models are the place to

- ▶ Connect to the database or any other data storage and external resource
- ▶ Implement CRUD (create, read, update, delete) methods
- ▶ Validate the data right before it actually goes into the data storage

#### Creating the Application Model

On our To Do List application, I want to save the data in a file. Note that using files as data storage is not recommended when talking about a real web application. Files tend to become locked pretty easily, which is not a good thing in this scenario.

The object we use for presenting the data is a simple hash with three values: ID, description, and creation date. I use the `Marshal` class for writing and reading it from a file.

Reading the file will be done when the model is loaded (happens when the model is needed, once per page load), which means that it occurs inside the constructor. The following code will be saved to `to_do_list_model.rb` file under the models directory:

```
class ToDoListModel
  # A constant with the data file path
  DataFileName = 'd:\temp\data.txt'

  def initialize
    if File.exist?(DataFileName)
      File.open(DataFileName, "r") do |f|
        @data = Marshal.load(f)
      end
    else
      @data = []
    end
  end
end
```

As you can see, the `ToDoListModel` class is a pure Ruby class with no superclass or mixin module that we have to add.

When we have the list ready, we can continue and add the public methods of the model. We need to provide three actions: `list`, `create`, and `delete`.

`list` is the simplest; it just returns the `@data` variable:

```
def list
  @data
end
```

The `create` method receives the description to add, generates its ID, and adds a new item to the hash. After that, it serializes the hash to the data file to keep the persistent data storage up-to-date:

```
def create_item(description)
  # Generate the new id
  id = if @data.nil? then 1 else @data.length + 1 end

  # Add the new record to the hash
  @data << { :id => id,
             :description => description.to_s,
             :creation_date => Time.now }

  # Save to file
  File.open(DataFileName, "w+") { |f| Marshal.dump(@data, f) }
end
```

For the `create` method, we also need to validate that the description is not empty. A to-do task without a description is like lemonade without lemons, isn't it?

This is why we add a validation method that we use later on in the controller:

```
def validate_for_creation(description)
  return false if description.to_s.length == 0
  # If data is valid, return true
  true
end
```

The `delete` method consists of removing the item with the given ID from the hash and then saving the new hash to the file:

```
def delete_item(id)
  # Delete the matching row from the hash
  @data.delete_if { |item| item[:id] == id }

  # Update the file
  File.open(DataFileName, "w+") { |f| Marshal.dump(@data, f) }
end
```

That's it! Listing 15.1 contains the code of the entire `ToDoListModel` class.

#### LISTING 15.1 The `ToDoListModel` Class

---

```
class ToDoListModel
  # A constant with the data file path
  DataFileName = 'd:\temp\data.txt'

  def initialize
    if File.exist?(DataFileName)
      File.open(DataFileName, "r") do |f|
        @data = Marshal.load(f)
      end
    else
      @data = []
    end
  end

  def list
    @data
  end

  def validate_for_creation(description)
    return false if description.to_s.length == 0
    # If data is valid, return true
    true
  end

  def create_item(description)
    # Generate the new id
    id = if @data.nil? then 1 else @data.length + 1 end

    # Add the new record to the hash
    @data << { :id => id,
               :description => description.to_s,
               :creation_date => Time.now }

    # Save to file
    File.open(DataFileName, "w+") { |f| Marshal.dump(@data, f) }
  end

  def delete_item(id)
```

```

# Delete the matching row from the hash
@data.delete_if { |item| item[:id] == id }

# Update the file
File.open(DataFileName,"w+") { |f| Marshal.dump(@data, f) }
end
end

```

---

## Controllers

Controllers are the input logic in an MVC application, and they are responsible for handling incoming web requests. They inherit, in most cases, from the `Controller` class.

Controllers expose actions. An action is a public controller method (it must be defined as public!) that gets invoked when a matching URL is visited.

Figure 15.6 shows the routing of a simple URL to a controller action.

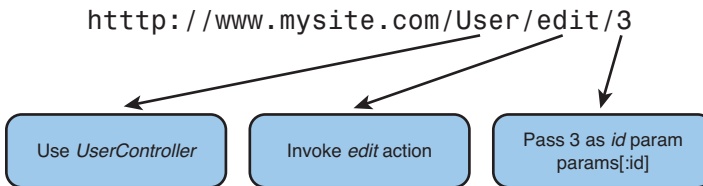


FIGURE 15.6 URL routing sample.

Controllers are the place to

- ▶ Interpret data of incoming requests
- ▶ Communicate with the model
- ▶ Decides which action to take (render a specific view, redirect to another controller, raise an error, and so on)

### Controller Features

Inside the controller, a few attributes enable you to provide information about the web request and response, as described in Table 15.1.

### Actions Return Values

Controller actions must return a value. This value represents the actual action that the MVC framework should take. The return type is `ActionResult` or one of its derivatives.

The commonly used `ActionResult`s are `ViewResult` and `RedirectResult`.

TABLE 15.1 Available Attributes Inside the Controller

| Attribute | Description   |
|-----------|---|
| Request   | An <code>HttpRequestBase</code> instance. Contains information about the web request. IronRubyMvc enhances this class and adds in more Ruby-like methods to investigate the request:<br><b>post?</b> , <b>get?</b> , <b>put?</b> , <b>delete?</b> , <b>head?</b> , <b>ajax?</b> |
| Response  | The HTTP response instance.   |
| Server    | An <code>HttpServerUtilityBase</code> instance. Contains the server name ( <code>machine_name</code> ) and methods for easier server related tasks (for example, <code>map_path</code> ).   |
| Session   | The session management class instance. Via the <code>Session</code> attribute, you can access session variables and configuration.  |

**ViewResult** This type of result instructs the ASP.NET MVC framework to send the given view as the response. The way to use it is by the IronRubyMvc `view` method, which constructs the `ViewResult` instance from the given parameters.

The `view` method retrieves three parameters:

- ▶ **View name (optional):** Specifies the name of the view to use. The default is decided according to the route table.
- ▶ **Master layout name (optional):** Specifies the name of the master layout to use. The default is none.
- ▶ **Model instance or a hash (optional):** This argument has two possible uses. One is to pass the model object to the view; the second is to pass a hash of data to the view. The view can then access the model or the hash via the `model` object.

The following code sample demonstrates different calling options to the `view` method:

```
# Use the default view according to the route table view

# Use index.html.erb view on the controller views folder
# (or the shared views folder), do not use a layout
# and do not send model data to the view
view 'index'
view :index # view name can be sent as a symbol as well

# Use list.html.erb view with a layout specified
# on main.html.erb
view "list", "main"
```

```
# Use index.html.erb view with no layout
# and pass it a MyModel instance
view "index", nil, MyModel.new

# Use about_me.html.erb view with main.html.erb layout
# and pass it a hash of data items
hash = { :name => "Shay Friedman",
         :blog => "http://www.ironshay.com" }
view 'about_me', 'main', hash
# Using this hash from the view will be done as follows:
# My name is <%= model[:name] %>
```

**RedirectResult** Sometimes you need to redirect the request to a different controller action. For example, after deleting a record, you would want to take the users to the list page instead of showing them a success message. This is what the `RedirectResult` is for.

To create this type of result, there are two helper methods on `IronRubyMvc` controllers: `redirect_to_action` and `redirect_to_route`. `redirect_to_action` is a more specific method than `redirect_to_route`. They both enable you to send the request to a different action method on the same controller, on another controller, or to a different route.

`redirect_to_action` takes two parameters: the action name and an optional route details hash. `redirect_to_route` receives only the route details hash.

The following sample code demonstrates different uses for these methods:

```
# Redirect to the index action method on the same controller
redirect_to_action "index"
# Same result using the redirect_to_route method:
redirect_to_route { :action => "index" }

# Redirect to the list action on the user controller
redirect_to_action "index", { :controller => "user" }
# or
redirect_to_route { :action => "index",
                   :controller => "user" }

# Redirect to the edit action on the same controller
# with an id parameter
redirect_to_action "edit", { :id => 5 }
# or
redirect_to_route { :action => "edit", :id => 5 }
```



There are other methods for other `ActionResult` responses, too:

- ▶ `json`: Responds with a JSON-formatted result. It is intended mostly for AJAX calls.
- ▶ `java_script`: Responds with a JavaScript script.
- ▶ `content`: Responds with a textual response.
- ▶ `file`: Responds with a file (can send the binary content, the path, or the file stream).

If you want to read more about them, refer to the ASP.NET MVC learning site at <http://www.asp.net/learn/mvc>.

### Method Selectors

Every controller method can be enhanced with selectors that change the way it works with the ASP.NET MVC framework.

There are three main selectors: `accept_verbs`, `non_action` and `alias_action`.

**accept\_verbs** This selector makes an action available only on a specific method: POST, GET, PUT, DELETE, or HEAD. This means that an action that is instructed to work on POST requests only will be entirely unavailable for other request methods like GET.

The following code shows how to use the `accept_verbs` selector method:

```
# Make the index method available for GET requests only
accept_verbs :index, :get
# Make the update method available for POST requests only
accept_verbs :update, :post
```

### USE ACCEPT\_VERBS REGULARY

A best practice for controller actions is to use `accept_verbs` for each and make the action respond only to its matching request types. For example, there is no need for an edit action that shows an edit form to receive POST requests, and there is no need for an update action that updates the database to receive GET requests.

The basic rule is to use `:post` for data-modifying actions and `:get` for data-reading actions.

**non\_action** As mentioned previously, public controller methods are considered action methods. Sometimes, however, you may want to have a public method but not want it to act as an action method. This is what the `non_action` selector is for. In such a case, just use the `non_action` selector method to instruct the MVC framework that your method is not an action method.

To use it, you just pass the method name as a symbol to the `non_action` selector method:

```
# Make calculate_something method a non-action method
non_action :calculate_something
```

### WHEN TO USE NON\_ACTION

Using the `non_action` method should be the last resort when you want to prevent a method from being considered as an action. The better way to do so in terms of code design is by making the method private or protected.

## alias\_action

When you need to provide a way to call a single action via different names, the `alias_action` can prove handy. For example, you might want the `list` action to be accessible via the `index` action. Writing the same code twice is bad for you, so using `alias_action` is the better alternative.

The first parameter to the `alias_action` selector method is the alias action name as a symbol, and the second argument is the actual action name as a symbol:

```
# Make list_alias an alias for the list action method
alias_action :list_alias, :list
```

With this line in the controller code, the next two URLs will be redirected to the `list` action (assuming it is written in the `UserController` class):

```
http://localhost/User/list
http://localhost/User/list_alias
```

## Using Libraries

If you make use of the standard library or some other third-party libraries and you don't have IronRuby installed on the computer (only the IronRubyMvc framework), copy the needed libraries to the `Libs` folder under the application root directory.

The `Libs` folder is one of the loaded paths of the application, so its content is available without having to specify the full path.

## Creating the Application Controller

For the To Do List application, we prepare a controller that takes care of the main actions on our page: listing, creating, and deleting.

The first task we complete is building the controller class and its constructor. The controller class will be named `ToDoListController`, and it will be saved to `Controllers\to_do_list_controller.rb`. Inside the constructor, we generate the model class to make it available and ready for the controller actions:

```
require "to_do_list_model"
class ToDoListController < Controller
  def initialize
    @model = ToDoListModel.new
  end
end
```

When the constructor is in place, let's move on to the action methods.

The first action is listing. This is a simple one. All we need is to get the data from the model, save it to an instance variable so that the view can use it, and call the view (with `to_do_layout` as its layout):

```
def index
  @data = @model.list
  view 'index', 'to_do_layout'
end
```

The action is now called `index` and we want it to be available as a list, too. To do so we use the `alias_action` method:

```
alias_action :list, :index
```

The next action is the `create` action. The `create` action has a page of its own with a form that contains the needed fields. The input can then be sent with the request to the `create` action:

```
def create
  return view("create", "to_do_layout") unless request.post?
  @model.create_item request.form['description']
  redirect_to_action 'list'
end
```

Notice that we act differently for POST requests. When a POST request is coming, it originates from the creation form, so we call the `create_item` method on the model and redirect to the `list` action (so that the user can see the new addition). Otherwise, we just show the creation form.

The last action is the `delete` action. This one does not have its own form and is invoked via a link on the index page. Inside the action method, we call the `delete_item` method on the model and then redirect to the `list` action:

```

def delete
  id].to_i)
  redirect_to_action 'list'
end

```

The `id` parameter is accessed via the `params` hash because it is one of the route parameters.

The `delete` action method is accessed via a link, so the only approved way to call the `delete` action is via a GET request. To enforce that, we use the `accept_verbs` selector:

```
accept_verbs :delete, :get
```

That sums up our work on the controller. Listing 15.2 contains the code of the entire `ToDoListController` class.

---

#### LISTING 15.2 The `ToDoListController` Class

---

```

require "to_do_list_model"

class ToDoListController < Controller
  def initialize
    @model = ToDoListModel.new
  end

  alias_action :list, :index

  def index
    @data = @model.list
    view 'index', 'to_do_layout'
  end

  def create
    return view("create", "to_do_layout") unless request.post?
    @model.create_item request.form['description']
    redirect_to_action 'list'
  end

  accept_verbs :delete, :get

  def delete
    id].to_i)
    redirect_to_action 'list'
  end
end

```

---

## Views

Views contain the UI logic. A view is connected to a specific action in the model. So, for every controller, you can find several different views: one for listing, one for creating new records, one for editing, and so forth.

Views are HTML files with IronRuby code inside: ERB files. ERB, which stands for eRuby, is a templating system that allows embedding Ruby code inside text documents. Views are located in subfolders of the Views folder. Each controller has a matching folder under the Views folder, which is where the controller-related views are placed.

Views are responsible for

- ▶ Rendering the HTML to show the model and controller data
- ▶ Implementing client behavior using JavaScript
- ▶ Using stylesheets
- ▶ Implementing view layouts

### Using Controller Data

Views can use data from the controller. This is done via the `view_data` object. Every instance variable of the controller class is available to the view through the `view_data` object. For example, if the controller contains the following statement

```
@title = "Title from the controller"
```

The view can use this data by calling `view_data.title`.

### View HTML Helper

When writing a view, you don't have to write the actual HTML by yourself. Instead, you can let the HTML helper do it. The helper, which is available via the `helper` object, contains several methods that aid in constructing HTML links to other controller actions and form-building tasks. Table 15.2 describes the main methods.

TABLE 15.2 Html Helper Methods

| Method                   | Description   |
|--------------------------|---|
| <code>action_link</code> | <p>Generates a link to another controller action. It retrieves two parameters: the display value; and a hash of controller name, action name, and parameters (each optional).</p> <p>For example:</p> <pre>&lt;%= html.action_link("Click me",                         {:controller =&gt; 'users'                          :action =&gt; 'show'                          :user_id =&gt; 5})</pre> |

TABLE 15.2 Html Helper Methods

| Method   | Description  |
|--|--|
| Encode   | Prepares a string to be shown on an HTML page (for example, replacing less-than signs with &lt;).<br>For example:<br><code>&lt;%= html.encode("&lt;problematic for HTML&gt;") %&gt;</code>   |
| begin_form<br>end_form   | Helps to create a form's begin and end tags. The begin_form can accept also the action name, controller name, parameters, and method type (POST <b>or</b> GET).<br>For example:<br><code>&lt;% html.begin_form({:action =&gt; "create"}) %&gt;</code><br><code>...form...</code><br><code>&lt;% html.end_form %&gt;</code>   |
| check_box<br>drop_down_list<br>hidden<br>list_box<br>password<br>radio_button<br>text_area<br>text_box | These create form elements. They receive the element name and value (optional).<br>For example:<br><code>&lt;%= html.text_box("name", "your name here") %&gt;</code><br><code>&lt;%= html.check_box("yes_no", true) %&gt;</code>   |
| validation_message<br>validation_summary   | Placeholders for validation messages (discussed in more detail later in this chapter). The validation_message is for a specific element, and validation_summary shows all the validation errors in the form.<br>For example:<br><code>&lt;%= html.text_box("name") %&gt;</code><br><code>&lt;%= html.validation_message("name","required") %&gt;</code><br><code>&lt;br/&gt;</code><br><code>&lt;%= html.validation_summary %&gt;</code> |

## HTML HELPER USAGE

The ASP.NET MVC HTML helper is there to, well, help you. The framework will not force you to use it. You are free to create the HTML by yourself if you find doing so more suitable for you.

Nonetheless, it is recommended to use the HTML helper because it automatically takes care of issues you would otherwise have to handle manually (for example, creating the correct URLs).

### Custom View Helpers

The HTML helper is a great helper, but it doesn't have all the answers. A day will come (it will!) when you find yourself with repeating UI logic across the application. Don't fear; just build a custom view helper.

Custom view helpers exist in the Helpers directory. The way to create a helper method is to open the `System::Web::Mvc::IronRuby::Helpers::RubyHtmlHelper` class and add methods to it.

Let's create a custom view helper. We call the method `link_google`, and it can help us to create links to search results on Google. We save this helper in a file called `google_helper.rb`:

```
module System::Web::Mvc::IronRuby::Helpers
  class RubyHtmlHelper
    def link_google(caption, query)
      "<a href=\"http://www.google.com/search?q=#{query}\">#{caption}</a>"
    end
  end
end
```

To use it in the view, we need to tell the view to load it. So as the first line of our selected view, we add the following line:

```
<% load "google_helper.rb" %>
```

With this line in our view, we can go on and use our new helper method:

```
<%= html.link_google("IronRuby Unleashed on Google",
  "IronRuby Unleashed") %>
```

## MAKING CUSTOM HELPERS AVAILABLE TO ALL

If you want to make your custom helper available to all views on the application, just add the load statement to the application master layout.

### Shared Views

In most web applications, you have a standard layout that can be permanent throughout the application. Shared views are exactly for that. Here you create ERB files that can be available for every controller. These files are located under `Views\Shared`.

Inside a shared view, to set the location to the page content, you should use the `<% yield %>` statement.

In our To Do List application, we want to use a single layout. The layout is based on the default ASP.NET MVC layout with a few modifications. Listing 15.3 contains the layout

code. Notice that most of the code is plain HTML, but I use the HTML helper there to create links to the application pages (which we will soon build, too). The layout file will be called `to_do_layout.html.erb`.

### LISTING 15.3 The To Do List Layout

---

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
  <title>IronRubyMvc To Do List</title>
  <link href="/Content/Site.css" rel="stylesheet" type="text/css" />
</head>
<body>
  <div class="page">
    <div id="header">
      <br/>
      <div id="title">
        <h1>To Do List</h1>
      </div>
      <div id="menucontainer">
        <ul id="menu">
          <li><%= html.action_link("List",
                                { :controller => "ToDoList",
                                  :action => "index" })%>
          </li>
          <li><%= html.action_link("Create",
                                { :controller => "ToDoList",
                                  :action => "create" })%>
          </li>
        </ul>
      </div>
    </div>
    <div id="main">

      <% yield %>

      <div id="footer">
      </div>
    </div>
  </div>
</body>
</html>
```

---



## MORE USES FOR SHARED VIEWS

The main use for shared views is as shared layouts (like we just did). However, shared views can also be used when you want a single view for multiple different controller actions.

As mentioned previously, views are located on a directory that matches the controller name. This is the most common case, but views can also be put in the shared views directory, and the ASP.NET MVC framework will look for them there, too.

### Creating the Application Views

With the knowledge we have now, we can go ahead and create our pages. We begin with the index page, which lists all the current to-do items.

The code includes IronRuby code (between the `<% %>` signs) that loops over the model data (handed in by the controller) and displays the tasks in a tabular layout. The code is saved as `index.html.erb` under the `views\ToDoList` directory:

```
<table>
<% view_data.data.each do |item| %>
<tr>
  <td><%= html.encode(item[:id]) %></td>
  <td><%= html.encode(item[:description]) %></td>
  <td><%= html.action_link("Remove",
                           {:action => "delete",
                            :id => item[:id]}) %></td>
</tr>
<% end %>
</table>
<% if view_data.data.length == 0 %> No tasks defined <% end %>
<p>
<%= html.action_link("Add new", {:action => "create"}) %>
</p>
```

As you can see, along with the Ruby loop, we're taking advantage of the HTML helper class to encode the items correctly and to add a Remove link. Also notice that we do not write any layout code; this is done by the shared layout we have created before.

The output page is presented in Figure 15.7.

We do not have any tasks currently, and we also don't have a view that enables us to add new ones. This is our next task: building the create action view.

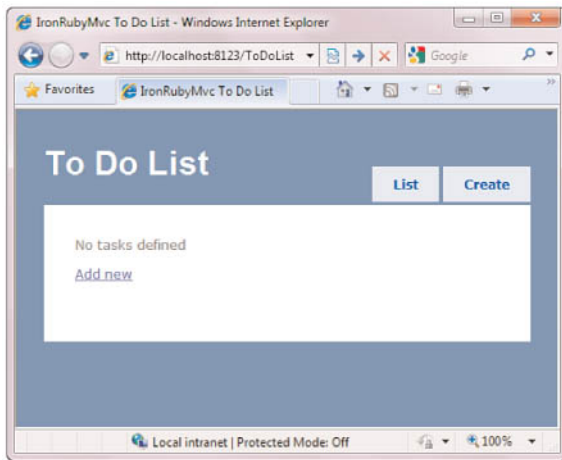


FIGURE 15.7 The list page without items.

In the create action view, we need a form where the user can input the task description and send it. We can use the HTML helper to create the form and form elements' HTML. The code will be saved to `create.html.erb` under `views\ToDoList`:

```
<h2>Add New</h2>
<% html.begin_form({:action =>"create"}) %>
  <label for="Description">Task description:</label>
  <%= html.text_box("description") %>
  <input type="submit" value="Create" />
<% html.end_form %>
```

If you wonder (or already frustrated) about the amount of HTML code this generates, you may be surprised to learn that the HTML code is as simple as it can be. This is the HTML that is generated by the preceding code:

```
<h2>Add New</h2>
<form action="/ToDoList/create" method="post">
  <label for="Description">Task description:</label>
  <input id="description" name="description"
        type="text" value="" />
  <input type="submit" value="Create" />
</form>
```

Figure 15.8 shows how the page turns out.

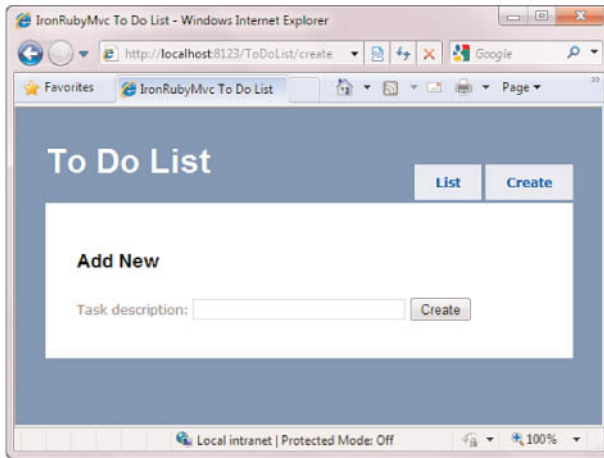


FIGURE 15.8 The new task page.

After adding a few tasks, the page looks like Figure 15.9.

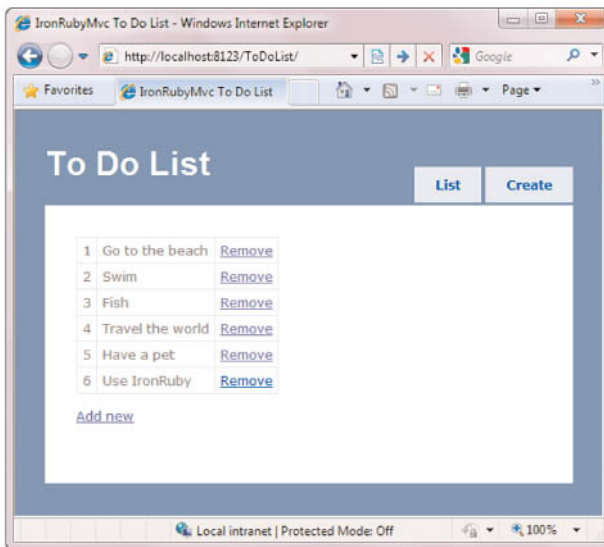


FIGURE 15.9 The list page with items.

The To Do List web application is now fully working—handling user input, reading and saving data from data storage, and looking good with a pleasant user interface. No doubt this is going to be the next big thing.

Even though the main functionality is written, there is more to the ASP.NET MVC framework. Throughout the rest of this chapter, we explore more capabilities and enhance our web application with their help.

## Routes

ASP.NET MVC routing is the first place incoming requests “meet” the application. The routing module is responsible to route the incoming request to its matching controller.

When you created the ASP.NET MVC application via the IronRubyMvc template, a routes file was added automatically. This file is located on the root of the application and has the “surprising” name of `routes.rb`.

This file already defines the default routing rule:

```
$routes.map_route("default", # Route name
                  "{controller}/{action}/{id}", # URL format
                  { :controller => 'Home',
                    :action => 'index',
                    :id => '' }) # Default values
```

The default route is a simple rule. For example, for the URL `users/delete/5`, the request will be sent to the `UserController` class, the `delete` action, with 5 as the `id` parameter.

Default values are also defined on the default route. In the preceding code, when the controller is not specified, the `Home` controller is used. When no action is specified, the `index` action is used. And when no `id` is specified, an empty string is used as the `id` value.

There is no problem with changing the default values. Not every application is expected to have a `HomeController` with an `index` action method.

I’d like to change the default to use the `ToDoListController`. You can also send more data items to the controller via the route. I will add an `item_amount` default value, which we use in the upcoming “Custom Routes” subsection:

```
$routes.map_route("default",
                  "{controller}/{action}/{id}",
                  { :controller => 'ToDoList',
                    :action => 'index',
                    :item_amount => 10,
                    :id => '' })
```

## Custom Routes

Aside from the default route, you can create your own routes to match your needs.

To add a new route, we add another row to the routes.rb file above the default route line (not doing so results in ignoring the custom route).

For example, the next route accepts URLs with a value representing the number of items to show:

```
$routes.map_route("with_item_amount",
  "LimitedList/{item_amount}",
  {:controller => 'ToDoList',
   :action => 'index',
   :item_amount => 10})
```

This custom route enables URLs as `/LimitedList/3`. It executes the `list` method on the `ToDoList` controller class with the given item number. If no number is given, `10` is used.

To support this new route on the `ToDoListController`, we change the `index` action method to send only the given number of items to the view:

```
def index
  @data = item_amount].to_i]
  view 'index', 'to_do_layout'
end
```

With this applied, you need to restart Cassini to refresh the route table. After that, calling `http://localhost/LimitedList/3` results in the list shown in Figure 15.10. (Remember we have six items in the data file.)

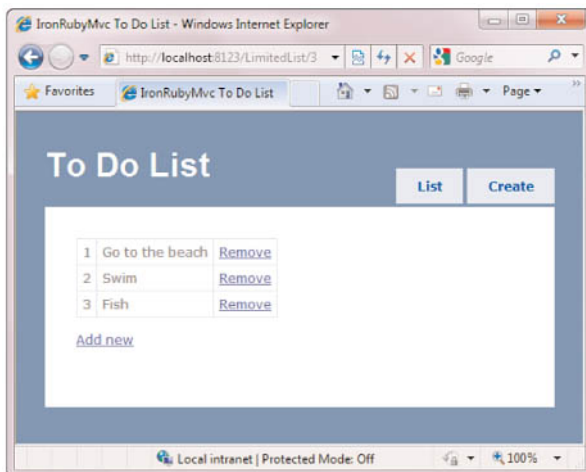


FIGURE 15.10 The index page showing a limited number of items.

## AVOIDING CUSTOM ROUTES WHEN POSSIBLE

For most small-to-medium web applications, custom routes are not necessary, and the default route can be sufficient. It is recommended to avoid custom routes when they are not needed. Using those changes the expected result of the application and as a result makes it harder to understand and debug.

## Filters

Filters are the ASP.NET MVC framework way of providing interference points for an incoming action request.

There are four types of filters:

- ▶ **Action filters:** Add code before and after an action is executed.
- ▶ **Result filters:** Add code before and after the result is sent back to the framework.
- ▶ **Authorization filters:** Add user authorization code. Can cancel the entire request if needed.
- ▶ **Exception filters:** The execution flow will get here when an exception is raised. You can then write code to log the error, show a nice error message to the user, or take any other action you want.

The different filters have different execution time frames. Figure 15.11 shows the flow of filters during an action execution.

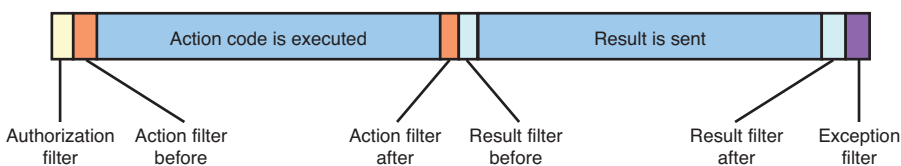


FIGURE 15.11 Flow of filters during an action execution.

## Action Filters

Action filters allow you to run code before and after an action is executed, but before the result is invoked.

Several action filter methods can be applied to a single action filter. For example, two methods are invoked on the `before_action` filter.

**before\_action**

In the “before” filter, you can investigate the request and change the result of the action. The `before_action` filter is supplied with a context variable of type `System.Web.Mvc.ActionExecutingContext`.

Table 15.3 describes the important attributes of the `ActionExecutingContext` object.

TABLE 15.3 Attributes of `ActionExecutingContext`

| Attribute Name                 | Description  |
|--------------------------------|--|
| <code>action_parameters</code> | Gets or sets the parameters passed to the action.  |
| <code>Controller</code>        | Gets or sets the controller instance. (Changing the controller here is not recommended.)   |
| <code>http_context</code>      | Accesses the HTTP context objects (session variables, application variables, cache, and more). Consists of the <code>HttpContextBase</code> class. |
| <code>route_data</code>        | Gets or sets the route data. You can access the route object and the values it passes.   |
| <code>Result</code>            | Gets or sets the result of the action. Note that when this attribute is set to a non-nil value, the action is canceled and won't be executed.      |

**CANCELING AN ACTION**

When you cancel an action (by setting the `Result` attribute to a non-nil value), all filters that are expected to be executed after the `before_action` method (for example, `after_action` or `before_result`) will not be executed. Only filter methods from other actions that should be executed (like in a case of action redirection) will be invoked as expected.

Using the `before_action` filter is done from inside the controller class. There are two ways of implementing it: via a block or via a method/lambda/proc. The following code uses the `before_action` method to write some request details to the response stream before the index action is executed:

```
before_action :index do |context|
  context.http_context.response.write("POST?
#{context.http_context.request.post?}")

  context.route_data.values.each do |val|
    context.http_context.response.write("<br/>#{val.key} = #{val.value}")
  end
end
```

An equivalent to the preceding code is as follows:

```
before_action :index, :write_context_details
def write_context_details(context)
  context.http_context.response.write "POST? #{context.http_context.request.post?}"

  context.route_data.values.each do |val|
    context.http_context.response.write "<br/>#{val.key} = #{val.value}"
  end
end
```

### after\_action

After the action is executed, the `after_action` filter methods are invoked. You cannot cancel the action here (because it has already run), but you can still change the result or investigate the request.

Like the `before_action` filter, the `after_action` one contains a context variable. This time it is of type `System.Web.Mvc.ActionExecutedContext`. It has the same attributes as the `ActionExecutingContext` except for the `action_parameters` attribute, which doesn't exist here, and a few additional attributes (described in Table 15.4).

TABLE 15.4 Additional Attributes of `ActionExecutedContext`

| Attribute Name    | Description  |
|-------------------|--|
| Cancelled         | Determines whether the action was canceled.  |
| Exception         | If an exception has occurred during the execution of the action, this attribute contains the exception object instance.  |
| exception_handled | Gets or sets a Boolean value indicating whether the exception was handled. If you do some exception handling here and you want to notify that the flow can continue, set this attribute to true. |

Using the `after_action` action filter is done from inside the controller via a block or a method. The next sample writes “action ended successfully” or “action ended unsuccessfully” to the response stream when the `index` action ends, according to the exception status:

```
after_action :index do |context|
  if context.exception.nil?
    msg = "action ended successfully"
  else
    msg = "action ended unsuccessfully"
  end
  context.http_context.response.write msg
end
```



**around\_action**

This action filter is a combination of the two actions previously described. When given a block, it executes the same block before and after the action (just make sure not to use unique context attributes when you do that):

```
around_action :index do |context|
  context.http_context.response.write "POST? #{context.http_context.request.post?}"

  context.exception # error! exception attribute does not exist on
                    # both before_action and after_action context objects
end
```

If you choose to pass lambdas or procs, it is done in a hash where you can pass the object for the `before_action` filter and for the `after_action` filter:

```
run_before = lambda { |context|
  context.http_context.response.write "BEFORE" }
run_after = lambda { |context|
  context.http_context.response.write "AFTER" }
around_action :index,
  { :before => run_before, :after => run_after }
```

**Result Filters**

Result filters are similar to action filters. The difference is the time of their execution. Result filters are executed after the action has finished running and before/after the result is invoked.

**before\_result**

This filter is executed before the result is invoked. This is the last place you can cancel the result. To do so, just set the `cancel` attribute to `true`. Other than canceling the result, you can investigate the request just as with the `before_action` and `after_action` filters.

The `context` parameter is a `ResultExecutingContext` object that contains the attributes described in Table 15.5.

We use it in a similar way to the way we have used the `before_action` filter: by passing it a block. The current version of `IronRubyMvc` does not support passing methods/procs/lambdaes to result filters:

```
before_result :index do |context|
  # Cancel PUT request
  if context.http_context.request.put?
    context.cancel = true
  end
end
```

TABLE 15.5 Attributes of ResultExecutingContext

| Attribute Name | Description   |
|----------------|---|
| Cancel         | Gets or sets a value indicating to cancel the result execution.   |
| Controller     | Gets or sets the controller instance. (Changing the controller here is not recommended.)  |
| http_context   | Accesses the HTTP context objects (session variables, application variables, cache, and more). Consists of the HttpContextBase class. |
| route_data     | Gets or sets the route data. You can access the route object and the values it passes.  |
| Result         | Gets or sets the result of the action. Notice that you can still change or replace the result object here.                            |

**after\_result**

On the `after_result` method, you can't cancel the result anymore (pretty obvious restriction). What you can do is check whether an exception has occurred and if it were taken care of, check whether the result was canceled, and access the web request information.

The `after_result` context variable is of type `ResultExecutedContext`. It does not contain a `cancel` attribute, but it has the extras additional attributes described in Table 15.6, which `before_result`'s `ResultExecutingContext` doesn't offer.

TABLE 15.6 Additional Attributes of ResultExecutedContext

| Attribute Name    | Description  |
|-------------------|--|
| Cancelled         | Determines whether the result was canceled.  |
| Exception         | If an exception has occurred, this attribute contains the exception object instance.   |
| exception_handled | Gets or sets a Boolean value indicating whether the exception was handled. If you do some exception handling here and you want to notify that the flow can continue, set this attribute to <code>true</code> . |
| Result            | Although it exists on <code>ResultExecutingContext</code> , too, it's a bit different. Here you can examine the result but not modify it (because it was already executed).                                    |

You use it with a block (methods/procs/lambda's are not supported for result filters currently):

```
after_result :index do |context|
  if context.canceled
    context.http_context.response.write "Result was canceled"
  end
end
```

**around\_result**

The `around_result` executes code before and after a result. It can receive a block that will be invoked (make sure not to use unique context attributes):

```
around_result :index do |context|
  context.http_context.response.write "Around the result"
end
```

Or you can use lambdas/procs:

```
do_before = Proc.new { |context|
  context.http_context.response.write "<br/>----->BEFORE"
}

do_after = Proc.new { |context|
  context.http_context.response.write "<br/>----->AFTER"
}

around_result :index, nil, { :before => do_before,
                             :after  => do_after }
```

**Authorization Filters**

Authorization filters are the first to be invoked. You want to set some special authorization rules inside them and cancel the request in case these rules are not followed.

The authorization filter is set via the `authorization_action` method, and the filter methods receive a context parameter of type `System.Web.Mvc.AuthorizationContext`.

The `AuthorizationContext` object contains the attributes described in Table 15.7.

TABLE 15.7 Attributes of `AuthorizationContext`

| Attribute Name | Description  |
|----------------|--|
| Controller     | Gets or sets the controller instance. (Changing the controller here is not recommended.)   |
| http_context   | Accesses the HTTP context objects (session variables, application variables, cache, and more). Consists of the <code>HttpContextBase</code> class.   |
| route_data     | Gets or sets the route data. You can access the route object and the values it passes.   |
| Result         | Gets or sets the result of the action. Note that when this attribute is set to a non-nil value, the action is canceled and won't be executed (neither will the other filters of the current action). |

The authorization filter is used with a block or a method, which is given to the `authorize_action` method. The following sample checks whether the current user exists in the Administrator role. If not, the result is set to a redirection result (to redirect the user to the login page):

```
authorized_action :index do |context|
  unless context.http_context.user.nil? and
    context.http_context.user.is_in_role("Admininstrator")
    context.result = System::Web::Mvc::RedirectResult.new("/account/log_on")
  end
end
```

The preceding code uses a block and is equivalent to the next one, which uses a method:

```
authorized_action :index, :auth
def auth(context)
  unless context.http_context.user.nil? and
    context.http_context.user.is_in_role("Admininstrator")
    context.result = System::Web::Mvc::RedirectResult.new("/account/log_on")
  end
end
```

## Exception Filters

Exception filters are the last to run. They are invoked only if an exception has occurred. Note that they run even if you have handled the exception on `after_action` or `after_result` filters and set the `exception_handled` attribute to `true`.

The exception filter is run after the result is invoked. This doesn't mean, however, that you're out of options. The ASP.NET MVC framework allows you to set the result one more time to show the user a nice error page.

Make sure to set the `exception_handled` to `true` if you really take care of the exception. Failing to do so can result in the event bubbling up and handled by a different exception handler (or the default one if none exist). If `exception_handled` is `false` or the new result ends up with an exception, the classic ASP.NET error pages are used (defined in the `customErrors` part in the `web.config` file).

The exception filter comes, like the others, with a context parameter. This time it is built of the `ExceptionContext` class. Its attributes are identical to some we've already seen (see Table 15.8).

To take advantage of exception filters, IronRubyMvc provides the `exception_action` filter. It receives a block and executes it when needed. Notice that methods, procs, and lambdas are not currently supported by this filter.

TABLE 15.8 Attributes of `HttpContext`

| Attribute Name                 | Description  |
|--------------------------------|--|
| <code>Controller</code>        | The controller instance.   |
| <code>http_context</code>      | Accesses the HTTP context objects (session variables, application variables, cache, and more). Consists of the <code>HttpContextBase</code> class. |
| <code>route_data</code>        | Gets or sets the route data. You can access the route object and the values it passes.   |
| <code>Exception</code>         | The exception object.  |
| <code>exception_handled</code> | Gets or sets a Boolean value indicating whether the exception was handled.   |
| <code>Result</code>            | Gets or sets the result of the action. After the result is set here, it will be executed.  |

The following sample code outputs some exception details to the response stream if the exception has already been handled:

```
exception_action :index do
  unless context.exception_handled
    context.http_context.response.write "An error has occurred!<br/>"
    context.http_context.response.write "Message: #{context.exception.message}"
  end

  context.exception_handled = true
end
```

## Controller-wide Filters

Until now, all the filters I've shown you are specific action filters. There is another possibility: making the filter controllerwide.

Controllerwide action filters run on their time slot for every action execution. Logging, error handling, and authorization—all of them are very likely to be needed for every controller action and not for a specific one.

To make a filter controllerwide, pass `nil` as the method name.

For example, the following code demonstrates some kind of lame logging mechanism that will be run before and after every controller action:

```
before_action nil do |context|
  context.http_context.response.write "Started on #{Time.now}"
end
```

```
after_result nil do |context|
  context.http_context.response.write "Ended on #{Time.now}"
end
```

## Custom Action Filter Classes

If you don't like the way of using filters that you've seen so far, you can do it differently by creating a filter class and adding it to the action filter list.

Every filter type has its own IronRuby filter class, which you can inherit from to implement your custom behavior.

Table 15.9 contains the class and method names. All classes are part of the module `System::Web::Mvc::IronRuby::Controllers`. (Remember to include it on the file you're using the filter classes.)

TABLE 15.9 FILTER SuperCLASSES and Methods

| Filter Type   | Class to Inherit From   | Method to Implement   |
|---------------|-------------------------|---|
| Action        | RubyActionFilter        | <code>on_action_executing(context)</code><br><code>on_action_executed(context)</code> |
| Result        | RubyResultFilter        | <code>on_result_executing(context)</code><br><code>on_result_executed(context)</code> |
| Authorization | RubyAuthorizationFilter | <code>on_authorization(context)</code>  |
| Exception     | RubyExceptionFilter     | <code>on_exception(context)</code>  |

To use your custom filter class, the Controller class contains a method named `filter`. This method received two parameters: a symbol representing the name of the action (`nil` for all actions), and the filter class instance.

Let's create a sample custom filter, an exception one. The class contains the `on_exception` method that runs the code we used earlier. The code can be added to the top of the `to_do_list_controller.rb` file. (Otherwise, the file should be loaded before using the filter.)

```
class MyException < System::Web::Mvc::IronRuby::Controllers::RubyExceptionFilter
  def on_exception(context)
    unless context.exception_handled
      context.http_context.response.write "An error has occurred!<br/>"
      context.http_context.response.write "Message: #{context.exception.message}"
    end

    context.exception_handled = true
  end
end
```

Now inside the controller, we can make all controller actions use this exception filter:

```
filter nil, MyException.new
```

That's it, you just implemented a custom filter.

Implementing the other filter types is exactly the same, just with a different superclass name and methods.

## Validations

On data-driven applications, you want to validate that users input expected values. The validation capabilities of ASP.NET MVC are meant to make this process easier.

### Inside the Model

No one knows the data and its expected format better than the model. That is why the validation itself should be done inside the model.

On our To Do List application, we add validation for the description field; we do not want it to be left blank.

We add a validation method to the model so that classes from other layers (someone say controllers?) can validate the data without needing to actually know what is expected from it:

```
def validate_for_creation(description)
  return false if description.to_s.length == 0
  # If data is valid, return true
  true
end
```

### Inside the Controller

The controller contains an object named `model_state`. This object is used by the view to determine whether the model action failed.

If the validation fails, the controller adds an error to the `model_state` object:

```
# The format: model_state.add_model_error(field_name, error)
model_state.add_model_error("description", "Description is required")
```

When an error is added to the `model_state`, the `model_state.is_valid` returns `false`, and the view knows something is wrong.

In our application, we change the create action to validate the data before adding it to the data storage:

```
def create
  return view("create", "to_do_layout") unless request.post?

  valid = @model.validate_for_creation request.form['description']
  unless valid
    model_state.add_model_error("description", "Description is required")
    view 'create', 'to_do_layout'
  else
    @model.create_item request.form['description']
    redirect_to_action 'list'
  end
end
```

The preceding code validates the data first. If it fails, it adds the validation error to the `model_state` object and shows the same page. If validation passes, the new record is created and the user is redirected to the `list` action.

## Inside the View

On the view side, there are some actions to be taken, as well. To make the user know about the error, view validation helpers come to our aid.

There are two validation helpers: single-field validator and validation summary.

The single-field validator, assuming you have a form field named `description`, is added as follows:

```
<%= html.validation_message("description", "required!") %>
```

This line could go near the field itself or anywhere else on the page.

The validation summary shows all the errors on the form in a bullet list, using the errors from the `model_state` object. The following line adds a validation summary to the view:

```
<%= html.validation_summary %>
```

With the validation elements, the To Do List application creation form consists of the following code:

```
<% html.begin_form({:action =>"create"}) %>
  <label for="Description">Task description:</label>
  <%= html.text_box("description") %>
  <input type="submit" value="Create" />
  <br/>
  <%= html.validation_message("description", "The description is required.") %>
<% html.end_form %>
```



If a user tries to create a to-do item with no description, he will be presented with the creation form again, but this time it will be presented with an obvious error notification, as shown in Figure 15.12.

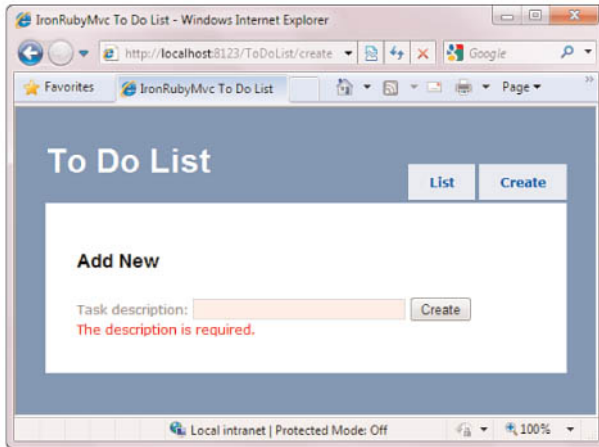


FIGURE 15.12 A validation error notification after an invalid input is submitted.

## Classic ASP.NET Features

Although ASP.NET MVC looks and feels very different from classic ASP.NET, it is still based on the classic framework. This means that all the features that were available to you in ASP.NET are also available to you in ASP.NET MVC.

It includes HTTP modules and handlers, session state management, authorization and authentications mechanisms, profiles, and more. Also notice that third-party components that have been built for classic ASP.NET still work in ASP.NET MVC views.

## A Word About Classic ASP.NET

Version 1.0 of IronRuby is not expected to support classic ASP.NET out-of-the-box, as known as Web Forms.

However, if you are interested in Web Forms support, keep an eye on the ASP.NET Dynamic Language Support project on CodePlex, <http://aspnet.codeplex.com/Wiki/View.aspx?title=Dynamic%20Language%20Support>.

This project is currently in its alpha version. It has support only for IronPython, but IronRuby support is coming soon.

## Summary

In this chapter, you learned about the ASP.NET MVC framework and how to use it with IronRuby. You learned about the MVC different parts, routes, action filters, validations, and classic ASP.NET capabilities.

With this knowledge, you can create your own web applications with IronRubyMvc.

This framework brings some fresh air to the world of ASP.NET. The community of developers who use it for their projects is growing by the minute, and you can find immediate help for any problem you encounter.

The best resource for ASP.NET MVC material is the official site: <http://www.asp.net/mvc>.

*This page intentionally left blank*

# CHAPTER 16

## Silverlight

In 2002, the term *RIA* was introduced. RIA, an acronym for rich Internet applications, is a generic name for applications that have the characteristics of desktop applications even though they are actually web applications. These applications differ from other web applications by their richness, which is expressed in an enhanced interface, media capabilities, and the overall “look and feel” of the application.

This rich interface cannot be achieved currently with the browser out-of-the-box capabilities like HTML and JavaScript. A special plug-in should be installed for that matter.

One of these plug-ins is the Silverlight plug-in. Silverlight started as a video streaming plug-in and gained more and more features as every version came out. The current version, Silverlight 3.0, is a mature framework that is used on popular websites. The most prominent use of Silverlight so far was for the Beijing Olympic Games, when the NBC Olympics website used Silverlight to give its users an incredible viewing experience. The site allowed users to rewind and watch a specific part again, watch live streams, and even watch in a picture-in-picture mode.

In this chapter, you learn what Silverlight is and how you can take advantage of it using IronRuby. I start by introducing Silverlight and its main concepts, then walk you through creating your first IronRuby-driven Silverlight application, and end with creating a small but fun Silverlight application.

### IN THIS CHAPTER

- Prepare Your Environment
- Hello, Silverlight
- Add Silverlight to a Web Page
- XAML
- Layout
- Controls
- Adding Code
- Graphics
- Media and Animations
- Data Binding

## Prepare Your Environment

The only piece of software you need to run Silverlight with IronRuby on your computer is the Silverlight 2.0 or 3.0 browser plug-in.

The plug-in can be obtained from the official Silverlight site at <http://silverlight.net>. This is the main site for Silverlight resources, so you might want to explore it a bit.

After you download and install the plug-in (and you have IronRuby on your system), you are ready to write IronRuby-driven Silverlight applications!

## Hello, Silverlight

Silverlight is a subset of Microsoft Windows Presentation Foundation (WPF) framework in terms of functionality and features. It provides graphics, multimedia, animations, and interactivity features. Moreover, a subset of the .NET framework is also available. The plug-in supports all major web browsers, including Internet Explorer, Firefox, and Safari, and can be run from Windows, Mac OS X, and Linux.

The concept of programming a Silverlight application is similar to a WPF application. Most of the layout, graphics, and animations are declared in XAML, and the code is written in IronRuby to support the layout and react to user actions.

Unlike WPF, where the user receives an EXE file with the program, here in Silverlight world we have a file called a XAP file (pronounced ZAP). This file is actually a zip file that contains the compressed assemblies and resources of the Silverlight application. The Silverlight plug-in then downloads the XAP file and runs it.

Creating your first Silverlight application takes just a few seconds. IronRuby comes with a handy tool to create everything for us, just run and play, and another one to run the application.

Let's get down to work and create our first Silverlight application.

### The sl Tool: The Silverlight Application Creator

The sl command-line tool is found in the IronRuby directory, within the silverlight\script folder.

This tool receives two arguments. The first one is the dynamic language you want to use: ruby or python. The second one is the path to where the application should be created:

```
sl [ruby | python] [application path]
```

Follow the next steps to create your first IronRuby-driven Silverlight application using the sl tool:

1. Open the command prompt. (Click Start > Run, type **cmd**, and press Enter.)
2. Navigate to the IronRuby installation folder. For example, if you have your IronRuby files on c:\IronRuby, enter the following:

```
cd c:\IronRuby
```

3. Navigate to the silverlight subfolder:

```
C:\IronRuby> cd silverlight
```

4. Now we use the sl tool and create an IronRuby-driven application in c:\SilverlightApps\IronRubySilverlight:

```
> script\sl ruby c:\SilverlightApps\IronRubySilverlight
5 File(s) copied

Your "ruby" Silverlight application was created in
c:\SilverlightApps\IronRubySilverlight\.
```

## SL TOOL TEMPLATES DIRECTORY

When you create a Silverlight application using the sl tool, it is created from a template that exists in the IronRuby directory under silverlight/scripts/templates/ruby. If you use the tool frequently and would like the template to contain other files, change the files in this directory.

### The Folder Structure

The basic template contains a few folders and files. Table 16.1 describes the different folders and their roles in the application.

TABLE 16.1 sl Tool Output Folder Structure

| Name       | Description  |
|------------|--|
| app        | Contains the application XAML and code files.        |
| css        | Contains the application stylesheet files.           |
| js         | Holds the application JavaScript files.              |
| index.html | The main page that runs the Silverlight application. |

## DEFAULT FOLDER STRUCTURE IS NOT A REQUIREMENT

The folder structure is not a requirement. It is how the basic template is built, and it gives you a good structure to begin with. Nevertheless, there is no problem with putting all files together on the same folder or on entirely different directories. It's up to you to decide.

## The chr Tool: The Development Server

The chr tool runs the dynamic language development server, Chiron (named after the intelligent centaur from Greek mythology).

Chiron dynamically packages everything you need to run the IronRuby-driven Silverlight application into a XAP file and serves it up to the browser. It also generates the required AppManifest.xaml file that contains information about the application assemblies and entry point.

chr has several command-line switches, as described in Table 16.2.

TABLE 16.2 chr Tool Command-Line Switches

| Switch                 | Description   | Sample  |
|------------------------|---|---|
| /w<br>or<br>/webserver | Launches the development server. It creates an XAP file automatically.<br>Port number can optionally be added to the switch, as well.   | Start the web server on default port 2060:<br>chr /w<br>Start the web server on port 3456:<br>chr /webserver:3456   |
| /b<br>or<br>/browser   | Starts the web server and launches the default browser. There is no need to use /w switch when this one is used.<br>Cannot be combined with /x or /z.<br>Start URL can optionally be added to the switch.                                     | Start the web server and default browser on the root folder:<br>chr /browser<br>Start the web server and launches the default browser on index.html:<br>chr /b:index.html |
| /z<br>or<br>/zipdlr    | Generates an XAP file, including dynamic language DLLs, and auto-generates AppManifest.xaml if it doesn't exist.<br>Doesn't start the web server.<br>Cannot be combined with /w or /b.<br>The file path must be added to the switch, as well. | Create a XAP file on my_file.xap:<br>chr /z:my_file.xap   |
| /m<br>or<br>/manifest  | Saves the generated AppManifest.xaml file to disk.<br>Doesn't start the web server.<br>Can only be combined with /d, /n, and /s.  | Save the manifest file to the current directory:<br>chr /m  |

TABLE 16.2 chr Tool Command-Line Switches

| Switch                               | Description  | Sample  |
|--------------------------------------|--|---|
| /d<br>or<br>/dir<br>or<br>/directory | Specifies the directory on disk. Default is the current directory.<br>The path must be added to the switch.  | Start the web server with files on c:\MyApp:<br>chr /w /d:c:\MyApp<br>Create the AppManifest.xaml file from sources found in c:\MyApp:<br>chr /m /d:c:\MyApp  |
| /r<br>or<br>/refpath                 | Path where assemblies are located. Default is the same directory as Chiron.exe.<br>The path must be added to the switch.   | Set the reference path to c:\IronRuby\bin:<br>chr /r:c:\IronRuby\bin  |
| /path                                | Adds paths to be included within the XAP file.<br>A semicolon-separated path string must be added.   | Create a XAP file and add c:\IronRuby and c:\IronPython to the paths:<br>chr /z:myfile.xap<br>/path:c:\IronRuby;c:\IronPython   |
| /x<br>or<br>/xap<br>or<br>/xapfile   | Creates a XAP file only with the directory content and without the manifest or the dynamic language DLLs.<br>Does not start the web server.<br>Cannot be combined with /w or /b.<br>File path must be added to the switch. | Create a XAP file for c:\MyApp and name it xap_file.xap:<br>chr /d:c:\MyApp<br>/xap:c:\MyApp\xap_file.xap<br>Create a XAP file for the current directory and name it xap_file.xap:<br>chr /xapfile:xap_file.xap |
| /n<br>or<br>/nologo                  | Suppresses display of the logo banner when starting the server.  | Start the web server on the current directory with no logo banner text:<br>chr /w /n  |
| /s<br>or<br>/silent                  | Suppresses display of all output.  | Start the web server on the current directory without any message being displayed:<br>chr /w /s   |

### Running Our First Silverlight Application

To run the Silverlight application we use the /b switch, which also starts the browser for us.

Open the command prompt. If you don't have IronRuby's Silverlight directory path on the Path environment variable, navigate to it. Otherwise, you can run the command from any other location. Now let's start the server:

```
> script\chr /d:C:\SilverlightApps\IronRubySilverlight /b:index.html
Chiron - Silverlight Development Utility. Version 1.0.0.0
Chiron serving 'c:\silverlightapps\ironrubysilverlight' as http://localhost:2060/
```



If `/s` or `/silent` wasn't used, these messages appear (`/n` or `/nologo` hides the first line). When requests are made to the server, the served files are written to the console, as well.

Because we started the server with the `/b` switch, the browser was also opened on the index page. There you find your first IronRuby-driven Silverlight application, as shown in Figure 16.1.

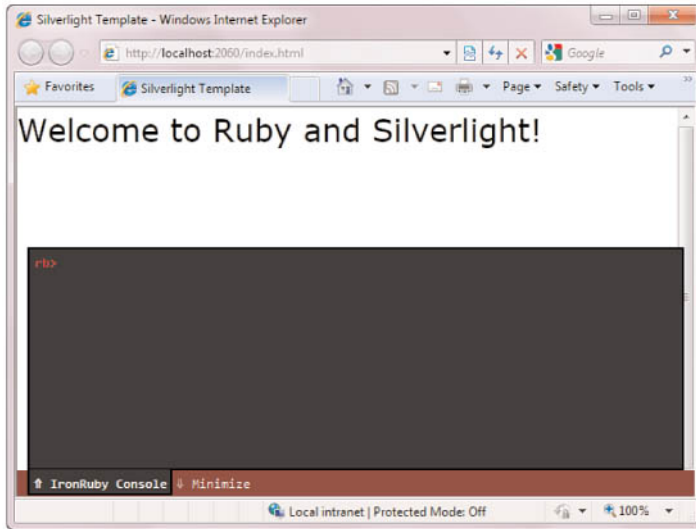


FIGURE 16.1 The default IronRuby-driven Silverlight page.

You probably wonder what the big black area is doing there. This is the REPL (read-evaluate-print loop) text area. It is added automatically by Chiron (can be called off, as well), and it allows you to write IronRuby code directly into the browser.

Go ahead and make a little statement in the browser; insert the next code line in the REPL area:

```
Application.current.root_visual.find_name("message").text = "IronRuby Rocks!"
```

Our statement and our page now look like Figure 16.2.

Now that we have the very basic Silverlight application, we can go on and enhance it to make it a real application (as far as *real* goes in book samples).

## Add Silverlight to a Web Page

Silverlight works in the web browser. As a result, it is loaded from the website HTML code. The HTML object element enables you to embed and configure the Silverlight plug-in:

```
<object data="data:application/x-silverlight,"
        type="application/x-silverlight-2"
        width="100%" height="100%">
  <param name="source" value="app.xap" />
</object>
```

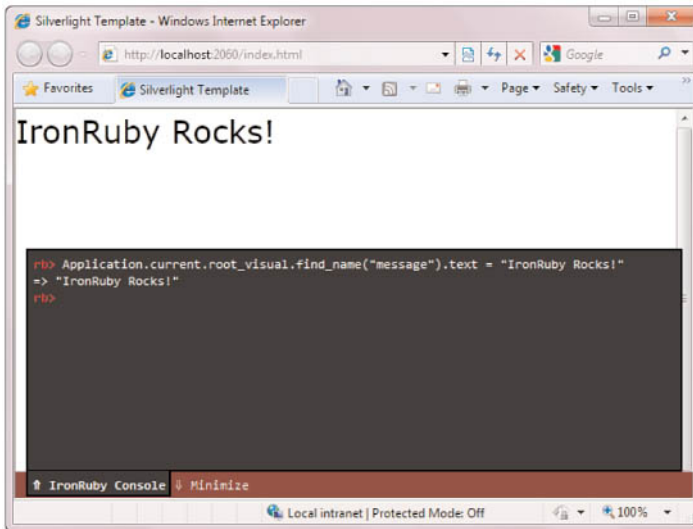


FIGURE 16.2 Using the IronRuby REPL console

The preceding code adds a Silverlight object to the page that fills the entire page. It is possible to provide different height and width values to make the object take only a portion of the page.

The source parameter is the most important parameter. It contains the name of the XAP file that holds the entire application. The source parameter is handled differently when you run the server via Chiron or via a different server (or even a simple HTML file). When Chiron is your server, the source parameter should point to the folder where the application start file (default is `app.rb`) is located. For example, in our application the `app.rb` file is located in the `app` folder. As a result, the source parameter value is `"app.xap"`. If the folder were named `my_app_folder`, the value would have been `"my_app_folder.xap"`. When you don't use Chiron as your server, the source parameter should point to a real XAP file.

The next important parameter for us is the one called `initParams`. The `initParams` contains a comma-separated key-value argument that enables passing user-defined initialization parameters to the Silverlight application. When you use Chiron, several unique keys are available for you within the `initParams` parameter, as described in Table 16.3.

The final parameter we discuss is the `onerror` parameter. When an error occurs and `initParams` doesn't contain an element ID for the error message, a JavaScript function, whose name is defined in the `onerror` parameter, is executed.

TABLE 16.3 initParams Chiron-Related Keys

| Key             | Available Value        | Description   |
|-----------------|------------------------|---|
| start           | Filename.rb            | The entry point of the application. The filename can be anything, but it must have the .rb extension for Chiron to know that we're working with IronRuby.<br>Default is app.rb.   |
| debug           | true or false          | Runs the application in debug mode. Stack traces will be presented in case of an error.<br>When it is set to true, you can attach the browser to Visual Studio debugger and debug your code (when Silverlight tools are installed).   |
| reportErrors    | ID of the HTML element | When an error occurs, the error message will be written to the innerHTML property of the HTML element with an ID matching the value of this key.<br>If there is no matching element with such an ID, an HTML element will be created with that ID, and the message will appear inside it.<br>If this key is omitted, no errors will be shown (unless the onerror parameter is defined). |
| exceptionDetail | true or false          | If set to true, it shows the entire managed stack trace rather than just the dynamic one.   |
| console         | true or false          | If set to true, the REPL console will be added to the bottom of the page.   |

For example, if the onerror parameter is declared as follows:

```
<param name="onerror" value="onSilverlightError" />
```

The JavaScript function that handles it can be as follows:

```
function onSilverlightError(sender, args) {
    alert(args.ErrorMessage);
}
```

You might want to look at the error.js file, which you can find within the js folder on the project directory. This is the default error handler and shows all the possible information about the error.

For more available parameters for the Silverlight object, visit the Silverlight Plug-in Object Reference at [http://msdn.microsoft.com/en-us/library/cc838259\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/cc838259(VS.95).aspx).

## XAML

XAML is a markup language that describes the UI of the Silverlight application. (WPF also uses it.) The UI elements are represented in the XAML code, including their properties. Moreover, XAML can contain static resources and animation definitions.

Our Silverlight application contains the default XAML file named `app.xaml` on the `app` folder. It contains a very simple page that allows showing the welcome message:

```
<UserControl x:Class="System.Windows.Controls.UserControl"
    xmlns="http://schemas.microsoft.com/client/2007"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid x:Name="layout_root" Background="White">
    <TextBlock x:Name="message" FontSize="30" />
  </Grid>
</UserControl>
```

If we want to add a blue rectangle to the page, we just add the following self-explanatory element right after the `TextBlock` element:

```
<Rectangle Height="70" Width="70" Fill="Blue" />
```

This makes our application look like Figure 16.3.

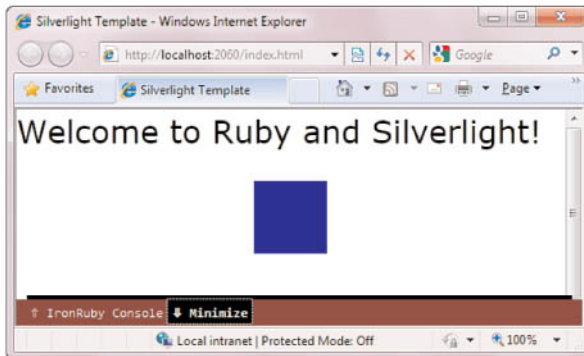


FIGURE 16.3 The Silverlight page with the new added rectangle.

To learn more about XAML, look at the “XAML” section in Chapter 13, “Windows Presentation Foundation (WPF).” Another good resource is the XAML Overview in the MSDN site: [http://msdn.microsoft.com/en-us/library/cc189036\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/cc189036(VS.95).aspx).

## Layout

In Silverlight, you cannot throw elements on a page without a layout element to contain them. The layout element provides the mechanism that is used to organize the elements. Table 16.4 describes the available layout controls for your Silverlight application.

TABLE 16.4 Silverlight Layout Controls

| Control    | Description  | Sample  |
|------------|--|---|
| StackPanel | Organize elements in a stack form—one on top of the other or side by side.<br>Good for simple needs. | <pre> &lt;StackPanel Orientation="Horizontal"&gt;     &lt;TextBlock&gt;Hello&lt;/TextBlock&gt;     &lt;Button Content="Push Me" /&gt; &lt;/StackPanel&gt; </pre>  |
| Grid       | Position elements in a grid form—by rows and columns.<br>Good for almost any layout need.            | <pre> &lt;Grid&gt;     &lt;Grid.ColumnDefinitions&gt;         &lt;ColumnDefinition Width="50" /&gt;         &lt;ColumnDefinition Width="*" /&gt;     &lt;/Grid.ColumnDefinitions&gt;     &lt;Grid.RowDefinitions&gt;         &lt;RowDefinition Height="100" /&gt;         &lt;RowDefinition Height="*" /&gt;     &lt;/Grid.RowDefinitions&gt;     &lt;TextBlock         Grid.Column="0"         Grid.Row="0"&gt;         Hello     &lt;/TextBlock&gt;     &lt;Button         Grid.Column="0"         Grid.Row="1"         Content="Push Me" /&gt;     &lt;Rectangle         Grid.Column="1" Grid.Row="0"         Grid.RowSpan="2" Height="70"         Width="70" Fill="Blue" /&gt; &lt;/Grid&gt; </pre> |

TABLE 16.4 Silverlight Layout Controls

| Control | Description  | Sample   |
|---------|--|--|
| Canvas  | Position elements in absolute positions—by providing its distance from the borders. Good for absolute positioning needs. | <pre> &lt;Canvas&gt;   &lt;TextBlock     Canvas.Top="10"     Canvas.Left="15"&gt;     Hello   &lt;/TextBlock&gt;   &lt;Button     Canvas.Top="30"     Canvas.Left="5"     Content="Push Me" /&gt; &lt;/Canvas&gt; </pre> |

The great thing about these layout controls is that you don't have to settle for only a single one. There is no problem with combining them all to satisfy your layout needs.

## Controls

Silverlight provides several different controls to help you build your application. The controls range from ones that show data, like `DataGrid`, `TreeView`, and `TextBlock`, to shape controls like `Rectangle`, `Ellipse`, and `Line`, to input controls like `TextBox`, `Button`, `DatePicker`, and more.

For a list and overview of all available controls, take a look at the MSDN page at [http://msdn.microsoft.com/en-us/library/cc645072\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/cc645072(VS.95).aspx).

## Adding Code

XAML is a wonderful markup language for views. However, XAML only is not enough when developing applications that should respond to user actions or execute complex tasks. For that you need to add code.

## Running XAML

With the default project template, you get the default code file, too. The file is `app.rb`, and it is located in the `app` folder. This file reveals to us the basic way to create a Silverlight application with IronRuby:

```
@root = Application.current.load_root_visual(UserControl.new, "app.xaml")
```

This code line creates the main Silverlight object from an XAML file. We can then access all elements via this object, remove them, or add new ones programmatically.

The `load_root_visual` method needs a bit of clarification. If you were to search for it on MSDN, you wouldn't easily find it. This is because it is not a built-in method of the Silverlight framework. It is part of the `Microsoft.Scripting.Silverlight` assembly, which is a set of extension methods that make some Silverlight tasks easier. The `load_root_visual` method does a simple trick: It uses the `load_component` method to load the XAML file and then sets the retrieved object to the `root_visual` property.

Note that the `root_visual` property can be set only a single time. As a result, there is no need to use the `load_root_visual` more than once in an application. If you need to load another XAML file, use the `load_component` method of the `Application` class.

The `load_component` method receives two arguments. The first one is an instance of an object of the same type as the root object of the XAML file. A `UserControl` would fit every need you have, so most of the time you pass `UserControl.new` as the first argument. The second argument is the path to the XAML file. The next sample code loads the XAML file `sl_file.xaml` into a variable named `sl_file_root`:

```
sl_file_root = Application.current.load_component(UserControl.new, "sl_file.xaml")
```

## REPLACING THE ENTIRE SILVERLIGHT CONTENT

As a result of `root_visual` being a one-time-only set attribute, we can't change the entire content in such a direct way as setting a new element to the `root_visual` attribute or using `load_root_visual` again.

If you still want to do it, there is a way. First, you can redirect the user to a different web page and have a new Silverlight content there. On the Silverlight way to replace the entire content, we need to remove all child nodes from the root element (which is one of the layout controls) and add the new root element to it. For example, assuming we have a new user control on a XAML file named `my_user_control.xaml`, all we have to do so that its content shows to the user rather than the current one is to run the following code:

```
new_root_object = Application.current.load_component(UserControl.new,
"my_user_control.xaml")
@root.children.clear
@root.children.add new_root_object
```

## Retrieving Silverlight Elements

The next obvious task after loading an XAML file is to access its elements. Unlike Windows or Web Forms, for which you have the elements as local variables, here you have to work a bit harder.

The way to retrieve an element is by the `find_name` method, which is available for every UI element in the Silverlight framework. The `find_name` method looks recursively into the element tree; so, by using the top element, you can use any element within the XAML page. The name it looks for is set to each element by the `x:Name` attribute.

For example, the next XAML user control contains two elements, a `StackPanel` named `layout_root` and a `CheckBox` named `my_checkbox`:

```
<UserControl x:Class="System.Windows.Controls.UserControl"
    xmlns="http://schemas.microsoft.com/client/2007"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <StackPanel x:Name="layout_root">
    <CheckBox x:Name="my_checkbox" Content="This is my checkbox" />
  </StackPanel>
</UserControl>
```

Assuming that the preceding XAML code is placed within a file named `sample.xaml`, this is how you can load the file, get the `CheckBox` object, and set it to be checked:

```
root_object = Application.current.load_component(UserControl.new, "sample.xaml")
checkbox = root_object.find_name("my_checkbox")
checkbox.is_checked = true
```

## MAKE RETRIEVING ELEMENTS EASIER USING RUBY CAPABILITIES

The process of retrieving an element in Silverlight is not as smooth as we're used to. Ruby has the tools to make this process much easier for us. By using the `method_missing` method, we can make the syntax much clearer and smoother. To use it, we open the `FrameworkElement` class, which is the base class for every Silverlight element:

```
include System::Windows
class FrameworkElement
  def method_missing(name)
    elem = self.find_name(name)
    if elem.nil?
      # The element is not found, continue with the original flow
      super
    else
      # Return the element to the caller
      elem
    end
  end
end
```



With this code available, instead of accessing an element in the regular way

```
@root.find_name("my_element")
```

we can write the following:

```
@root.my_element
```

## Event Handling

One of the most important tasks when writing an application is to respond to user actions. In Silverlight, when an action is executed, an event is raised. For example, when a button is clicked, the button element `click` event is invoked. All we've got left to do is to subscribe to the event we are after and respond in a way that fit our needs.

For example, if we have the preceding XAML file with the check box inside, we can subscribe to the check box's `click` event and react when the user changes its state:

```
my_checkbox = root_object.find_name("my_checkbox")
my_checkbox.click do |sender, args|
  my_checkbox.content =
    "This is my checkbox (current value: #{my_checkbox.is_checked.to_s})"
end
```

## Accessing the HTML Page and Window

One of the great things about Silverlight is its capability to interact with the browser. Silverlight contains an HTML bridge—managed classes that make it very easy to interact with the browser page (including JavaScript code) and window without leaving the comfort of IronRuby.

The main object for HTML interaction is `HtmlPage`. It is part of the `System::Windows::Browser` namespace. This is a CLR static object, so there is no need to initialize it. Via this object, we can access the document HTML, browser details, window, JavaScript, and more.

The following sample code creates a string message with some browser and page information and shows it in a JavaScript alert box:

```
include System::Windows::Browser
str = <<DOC
  Browser: #{HtmlPage.browser_information.name}
  Operating system: #{HtmlPage.browser_information.platform}
  URL: #{HtmlPage.document.document_uri.to_s}
DOC
HtmlPage.window.alert str
```

Figure 16.4 shows the sample output of the preceding code.

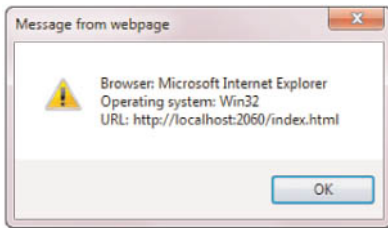


FIGURE 16.4 Showing a JavaScript alert box via IronRuby code.

For more information about the Silverlight HTML bridge, visit the related MSDN section at [http://msdn.microsoft.com/en-us/library/cc645076\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/cc645076(VS.95).aspx).

## Graphics

To this point, we covered several Silverlight features (but nothing you couldn't have accomplished with just plain HTML). Now we start to examine the great power and innovation in Silverlight.

Silverlight provides several different visual effects that you can create in XAML code (or by dynamically create them via code): drawings, shapes, paths, and complex geometries.

For example, the house drawing in Figure 16.5 is done entirely in XAML code.

The XAML code to draw it is as follows:

```
<Canvas>
  <Rectangle Canvas.Left="88" Canvas.Top="80" Height="70"
    Width="90" Stroke="Black" />
  <Polyline Canvas.Left="88" Canvas.Top="50"
    Stroke="Black" Points="0,30 45,0 90,30" />
  <Ellipse Canvas.Left="95" Canvas.Top="85" Height="22"
    Width="23" Stroke="Black" />
  <Ellipse Canvas.Left="145" Canvas.Top="85" Height="22"
    Width="23" Stroke="Black" />
  <Rectangle Canvas.Left="119" Canvas.Top="119"
    Height="31" Width="26" Stroke="Black" />
</Canvas>
```

Apart from these graphics capabilities, every Silverlight element can be filled with solid colors, color gradients, images, or even video clips. These effects are done with Silverlight brushes. Let's add some color to our house with the different brushes, as shown in Figure 16.6.

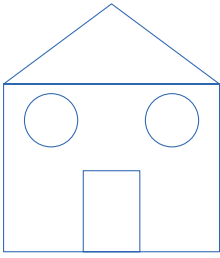


FIGURE 16.5 A house drawing created by XAML.

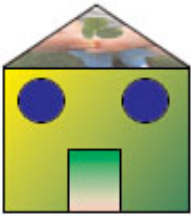


FIGURE 16.6 The house drawing filled with different brushes.

Want to draw this crazy house yourself? Well, I'm sure you do. The following XAML code, which differs from the previous one only by its use of brushes, will draw the house:

```
<Canvas>
  <!-- House structure with a gradient brush - >
  <Rectangle Canvas.Left="88" Canvas.Top="80" Height="70"
    Width="90" Stroke="Black">
    <Rectangle.Fill>
      <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
        <GradientStop Color="Yellow" Offset="0"/>
        <GradientStop Color="Green" Offset="1"/>
      </LinearGradientBrush>
    </Rectangle.Fill>
  </Rectangle>
```

```

<!-- Roof with an image brush - >
<Polyline Canvas.Left="88" Canvas.Top="50"
    Stroke="Black" Points="0,30 45,0 90,30">
    <Polyline.Fill>
        <ImageBrush ImageSource="flower.jpg" />
    </Polyline.Fill>
</Polyline>

<!-- Left window with default brush (solid color brush) - >
<Ellipse Canvas.Left="95" Canvas.Top="85" Height="22"
    Width="23" Stroke="Black" Fill="Blue" />

<!-- Right window with solid color brush - >
<Ellipse Canvas.Left="145" Canvas.Top="85" Height="22"
    Width="23" Stroke="Black">
    <Ellipse.Fill>
        <SolidColorBrush Color="Blue" />
    </Ellipse.Fill>
</Ellipse>

<!-- Door with gradient brush - >
<Rectangle Canvas.Left="119" Canvas.Top="119"
    Height="31" Width="26" Stroke="Black">
    <Rectangle.Fill>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
            <GradientStop Color="Lime" Offset="0" />
            <GradientStop Color="Pink" Offset="1" />
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
</Canvas>

```

To read more about brushes, visit [http://msdn.microsoft.com/en-us/library/cc189003\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/cc189003(VS.95).aspx).

In addition to brushes, Silverlight supports 2D and 3D transforms. For example, you can rotate an image 45 degrees to the right, or you can rotate it in 3D to add deepness to the UI.

For more information about transforms and 3D transforms, visit the relevant MSDN page at [http://msdn.microsoft.com/en-us/library/cc189037\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/cc189037(VS.95).aspx) and [http://msdn.microsoft.com/en-us/library/dd470131\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/dd470131(VS.95).aspx).

## Media and Animations

Silverlight offers video, audio, and animation capabilities. These capabilities are built in and are easy to integrate into the pages.

Adding a video is done with the `MediaElement` element. Just provide it with the URL to the movie, the height and width, and you're good to go:

```
<MediaElement Source="Wildlife.wmv" x:Name="movie" Height="300" Width="300"/>
```

`MediaElement` provides a way to control it via code. It supports three methods: `play`, `pause`, and `stop`. For example, if we want to pause the movie, we can use the next line of code:

```
@root.find_name('movie').pause
```

To read more about video and audio in Silverlight, navigate to [http://msdn.microsoft.com/en-us/library/cc189078\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/cc189078(VS.95).aspx).

Besides media capabilities, Silverlight has great animation capabilities. Every element can be animated. We can even animate a check box, for instance. The animation in Silverlight is based on changing an element property value over time. For example, `DoubleAnimation` changes a property of CLR type `Double`, `Int32Animation` changes a property of CLR type `Int32`, and so on. That means that to fade an element in and out, we need to use a `DoubleAnimation` that changes the value of its `Opacity` property from `0.0` to `1.0`.

The animation goes into a `Storyboard` element. This element is an animation container. You can set the time when each animation takes place, and you can even nest storyboards.

When you have a storyboard, you need to define when the animation begins. This is done by associating the storyboard with an event. The association is done entirely in XAML code and does not require IronRuby code (even though it can be defined via code).

The next XAML code makes the Hello text grow and shrink infinitely. The animation starts when the `TextBlock.Loaded` event is triggered:

```
<TextBlock Name="txt" FontSize="1">Hello
  <TextBlock.Triggers>
    <EventTrigger RoutedEvent="TextBlock.Loaded">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation
            Storyboard.TargetName="txt"
            Storyboard.TargetProperty="FontSize"
            From="1.0" To="60.0" Duration="0:0:2"
            AutoReverse="True" RepeatBehavior="Forever" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </TextBlock.Triggers>
</TextBlock>
```

For more information about Silverlight animations, visit [http://msdn.microsoft.com/en-us/library/cc189019\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/cc189019(VS.95).aspx).

# Data Binding

In addition to looking good, Silverlight knows how to display data in a convenient way. Several data binding options and controls are available for you: static data binding, dynamic data binding, and data templates.

## Static Data

Sometimes there is a certain data item that you want to store in a single location. This way you can use it everywhere you need it without the need to duplicate it. This is called a resource in the XAML world. Every element has its resource storage, and it is available for itself and all its child elements.

For example, we can store a brush we'd like to use across the page. Then we can just refer to the resource whenever we need it instead of defining the brush explicitly. The following XAML code does exactly that. The brush is saved on the `StackPanel` resource storage (`StackPanel.Resources`) and is used on all elements inside it. Note that to identify a resource, we give it a name using the `x:Key` attribute:

```
<StackPanel>
  <StackPanel.Resources>
    <SolidColorBrush x:Key="olive_color" Color="Olive" />
  </StackPanel.Resources>
  <TextBlock Foreground="{StaticResource olive_color}">Hello</TextBlock>
  <Rectangle Height="50" Width="50" Fill="{StaticResource olive_color}" />
</StackPanel>
```

## Styles

One of the more useful methods of static resources is styles. Silverlight provides a way to define a whole set of properties. After this set is applied to an element, all the properties are applied to it. This is an easy way to create a style for the application.

The style definition is declared within the `Resources` tag. It has two important attributes: `x:Key` to define its name, and `TargetType`, which is the name of the type the style will be applied to. Inside the style definition setters of attribute name and value will be declared. The next style is supposed to be applied on `TextBlock` elements and set its foreground color and text:

```
<StackPanel.Resources>
  <Style x:Key="the_ironruby_style" TargetType="TextBlock">
    <Setter Property="Foreground" Value="Red" />
    <Setter Property="Text" Value="IronRuby!!!" />
  </Style>
</StackPanel.Resources>
```

To apply this style on a `TextBlock` element, we use the `StaticBinding` declaration for the element `Style` attribute:

```
<TextBlock Style="{StaticResource the_ironruby_style}"/>
```

Besides styles, there are also control templates, which make it possible for you to create a whole element tree as a template. To read about control templates, take a look at [http://msdn.microsoft.com/en-us/library/system.windows.controls.controltemplate\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/system.windows.controls.controltemplate(VS.95).aspx).

## Dynamic Data

Most of the data binding code you write will probably be related to dynamic data. Silverlight supports binding to data that comes from code. However, you can bind only to reflection-based properties, which is not the case with IronRuby (it uses `ICustomTypeDescriptor` instead).

The way to work around the problem is to use CLR objects. There is no problem filling them with IronRuby; the only restriction is that they not be native IronRuby objects.

To bind to dynamic data, we first create a CLR object. The next VB.Net class will be contained in `country.dll` as part of the `CLRObjects` namespace:

```
Public Class Country
    Private _name As String
    Private _capitalCity As String

    Public Property Name() As String
        Get
            Return _name
        End Get
        Set(ByVal value As String)
            _name = value
        End Set
    End Property
    Public Property CapitalCity() As String
        Get
            Return _capitalCity
        End Get
        Set(ByVal value As String)
            _capitalCity = value
        End Set
    End Property

    Public Sub New(ByVal name As String, ByVal capitalCity As String)
        _name = name
```

```

    _capitalCity = capitalCity
End Sub
End Class

```

Our next task is to create the data in IronRuby:

```

require "country.dll"
united_states = CLRObjects::Country.new("USA", "Washington D.C")

```

If we want to use this data and present it to the user, we need to set the data context of the target element with our data object and make sure the element attributes and child elements are bound correctly to the expected data.

In our sample, we bind the data to TextBlock objects that show the country name and capital city. The following XAML code does this binding:

```

<StackPanel x:Name="layout_root">
  <TextBlock>Country name:</TextBlock>
  <TextBlock Text="{Binding Name}" />
  <TextBlock>Capital city:</TextBlock>
  <TextBlock Text="{Binding CapitalCity}" />
</StackPanel>

```

The last piece of the puzzle is to actually set the data object as the context of the page. The `layout_root` StackPanel is the root of the application, so we already have its object (and we don't have to find it). The following IronRuby code loads the form and sets the data object to the StackPanel. Note that when you data bind an element, all of its subelements are also getting bound to that data:

```

@root = Application.current.load_root_visual(UserControl.new, "app.xaml")
united_states = CLRObjects::Country.new("USA", "Washington D.C")
@root.data_context = united_states

```

That's it. Figure 16.7 shows the output of our application.

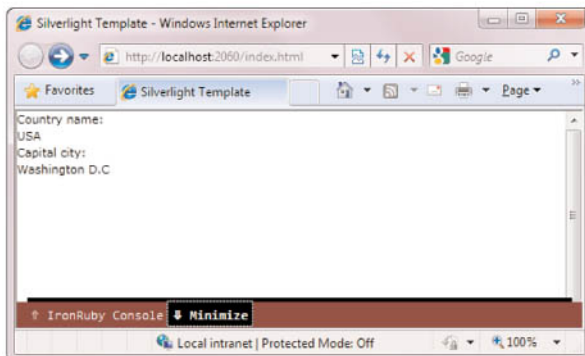


FIGURE 16.7 A Silverlight page showing dynamic data.



## Data Templates

In addition to binding to a single data item, Silverlight supports data binding to a list of data items. This is done using data templates—a template where you define the way a single item will be presented. This data template is then used for each item on the list.

For example, let's assume we have an array (there is no problem the array is native IronRuby) of Country objects:

```
countries = [CLRObjects::Country.new("USA", "Washington D.C"),
             CLRObjects::Country.new("England", "London"),
             CLRObjects::Country.new("Ireland", "Dublin")]
```

On the XAML side, we create a combo box to show them. In the combo box definition, we add a data template that defines how each item on the list is shown on the list:

```
<Canvas>
  <ComboBox x:Name="combo" ItemsSource="{Binding}">
    <ComboBox.ItemTemplate>
      <DataTemplate>
        <StackPanel Orientation="Horizontal">
          <TextBlock Text="{Binding Name}"/>
          <TextBlock Text=" (" />
          <TextBlock Text="{Binding CapitalCity}"/>
          <TextBlock Text=")"/>
        </StackPanel>
      </DataTemplate>
    </ComboBox.ItemTemplate>
  </ComboBox>
</Canvas>
```

With the data template in place, all we have left to do is to set the countries array as the data context of the combo box:

```
@root.find_name("combo").data_context = countries
```

Figure 16.8 shows the output.

Data templates are used massively within data-driven applications. They appear in grid controls, menu controls, and any other list-related control. Even though binding IronRuby objects to Silverlight controls is unavailable at the moment, the workaround is bearable and allows you to use IronRuby to write your Silverlight applications almost seamlessly.

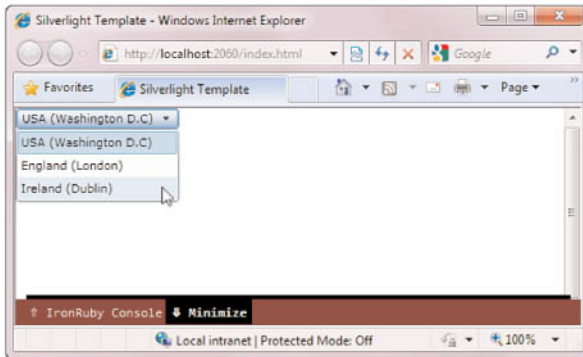


FIGURE 16.8 A Silverlight combo box showing data via data templates.

## Summary

In this chapter, you learned the fundamentals of Silverlight and IronRuby. You learned about the tools IronRuby has for Silverlight, the XAML language, the different controls Silverlight offers, graphics and animations, and data binding—everything by using the IronRuby language. There is a lot more in Silverlight, and if you find it interesting, I recommend that you peruse the MSDN library or Silverlight-specific books.

Silverlight is becoming more and more relevant as more and more websites take advantage of it.

Moreover, as a result of the easy integration between Silverlight and HTML, we're starting to see some interesting projects that might affect the entire client development field for web application. One of the most impressive projects is the Gestalt project. By using Silverlight behind the scenes, the Gestalt project allows writing Ruby and Python scripts rather than JavaScript. For example, the following sample shows an alert box with a welcome message according to user input:

```
<html>
<head>
    <script src="js/jquery.js" type="text/javascript"></script>
    <script src="js/gestalt.js" type="text/javascript"></script>
</head>
<body>
<input id="username" type="text"/>
<input id="say_hello" type="button" value="Say, Hello!" />
```

```
<script language="ruby">
document.say_hello.onclick do |s,e|
    window.alert "Hello, #{document.username.value}!"
end
</script>
</body>
</html>
```

For more information about the Gestalt project, visit the project home page at <http://www.visitmix.com/labs/gestalt>.

# CHAPTER 17

## Unit Testing

Software quality is an important subject. Improving software quality is a big task, which includes code reviews, team meetings, code quality tools, QA, and more. The one task that you, as a developer, can do to make sure your code works as expected even before it leaves your development machine is to test it. By that, I don't mean to run it once, see that it works, and move on. I mean to write unit tests for it.

A unit test is a piece of code that runs the software and tests its output. Usually a unit test tests the smallest portion of code it can. For example, every method will be tested to make sure it works as expected. By having code that tests the application code, you gain a lot for “free.” For example, the code is more stable. When you make changes to the application code, you can always retest it to see whether the changes affected the expected result. And, unit tests also act as informal documentation of the code. (By reading the test code, you understand how to use the method and what it does.)

There are a lot of different unit testing frameworks. In this chapter, I show you three common Ruby unit testing frameworks: `Test::Unit`, `RSpec`, and `Cucumber`. We use these frameworks to test CLR code. It's having the best of both worlds: Continue coding the application with the static language of your choice, and test it using IronRuby while taking advantage of its amazing syntax and capabilities.

### IN THIS CHAPTER

- ▶ The Tested Code
- ▶ `Test::Unit`
- ▶ `RSpec`
- ▶ `Cucumber`

## The Tested Code

In this chapter we test a specific CLR class named `Calculator`. The calculator is not a regular one, it is a numerological calculator that calculates the numeric value of a given string according to Numerological rules.

List 17.1 contains the class code. If you want to use it, just create a new Class Library project, copy the code there, and build it.

LISTING 17.1 The `Numerology.Calculator` class

---

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Numerology
{
    public class Calculator
    {
        public Calculator()
        {
        }

        public int GetStringNumerologicalValue(string str)
        {
            int total = 0;
            foreach (char c in str.ToLower())
            {
                total += GetCharValue(c);
            }

            return SummarizeDigits(total);
        }

        private int GetCharValue(char c)
        {
            if (c == 'a' || c == 'j' || c == 's') return 1;
            if (c == 'b' || c == 'k' || c == 't') return 2;
            if (c == 'c' || c == 'l' || c == 'u') return 3;
            if (c == 'd' || c == 'm' || c == 'v') return 4;
            if (c == 'e' || c == 'n' || c == 'w') return 5;
            if (c == 'f' || c == 'o' || c == 'x') return 6;
            if (c == 'g' || c == 'p' || c == 'y') return 7;
            if (c == 'h' || c == 'q' || c == 'z') return 8;
            if (c == 'i' || c == 'r') return 9;
        }
    }
}
```

```

    throw new NotSupportedException("Char value not supported");
}

private int SummarizeDigits(int num)
{
    // The way to summarize the number's digits is to take the value modulo 9.
    // If 0 is the result, 9 should be returned.
    int sum = num % 9;
    if (sum == 0) return 9;
    return sum;
}
}
}

```

---

## Test::Unit

The Ruby language comes with a built-in testing framework available under the `Test` module. If you're familiar with .NET's NUnit, you will feel at home here. The `Test` unit testing framework is built of test suites, cases, methods, and eventually and most important, assertions.

To use the test framework, you need to require `'test/unit'` first. So the following line of code should be added to every file you write tests in:

```
require 'test/unit'
```

To write a test, you need to create a class that contains the tests.

### Test Cases

The class should inherit from the `Test::Unit::TestCase` class. This way, you point the test framework that this class is a `Test` class. We create a `Test` class named `TC_NumerologicalCalculatorPublicTest`:

```
class TC_NumerologicalCalculatorPublicTest < Test::Unit::TestCase
end
```

### NAMING CONVENTIONS

There is a recommended naming convention for the unit testing framework:

- ▶ Test case class names should be preceded with `TC_`.
- ▶ Test suite class names should be preceded with `TS_`.

Tests method must start with `test_`, so this convention is pretty much forced on you and is not just a recommendation.

---

Now we should add some test methods. For a method to be treated as a test method, its name must start with `test_`.

We write two tests—one that calculates a string and one that calculates a string with a single character (remember to also require the CLR assembly):

```
require 'test/unit'
require "NumerologicalCalculator.dll"
class TC_NumerologicalCalculatorPublicTest < Test::Unit::TestCase
  def test_string
    instance = Numerology::Calculator.new
    result = instance.get_string_numerological_value "Shay"
    assert_equal 8, result
  end

  def test_single_character
    instance = Numerology::Calculator.new
    result = instance.get_string_numerological_value "a"
    assert_equal 1, result
  end
end
```

## TESTING PRIVATE METHODS

IronRuby provides an easy way to access private CLR methods by using the `-X:PrivateBinding` switch. With private binding mode on, you can also test private CLR methods as if they were public. There is no need to code the test methods differently. Just run the tests with the `-X:PrivateBinding` switch.

Note that testing private methods is not recommended since it tightly couples tests to internal details of classes.

---

Note that every test method should have an `assert` method call. Without an assertion, the test is useless because it doesn't verify anything, and as a result, it never fails.

## Assertions

There are several different types of `assert` methods in the `Test::Unit` framework, as described in Table 17.1.

TABLE 17.1 Assert Methods

| Method   | Description   |
|--|---|
| <code>assert(boolean, message)</code>                          | <p>Passes if the given Boolean value is true. Otherwise, the test fails.</p> <p>For example:</p> <pre>assert 6 == "IronRuby".size</pre>   |
| <code>assert_equal(expected, actual, message)</code>           | <p>Passes if the expected and the actual values are equal.</p> <p>For example:</p> <pre>assert_equal 6, "IronRuby".length</pre>   |
| <code>assert_not_equal(expected, actual, message)</code>       | <p>Passes if the expected and the actual values are not equal.</p> <p>For example:</p> <pre>assert_not_equal 0, "Shay".length</pre>   |
| <code>assert_block(message) { }</code>                         | <p>Passes if the given block returns true.</p> <p>For example:</p> <pre>assert_block {   6 == "IronRuby".length }</pre>   |
| <code>assert_in_delta(expected, actual, delta, message)</code> | <p>Passes if the expected and actual values are equal with delta tolerance. Works on float values only.</p> <p>For example:</p> <pre>assert_in_delta(5.0, 4.7, 0.3)</pre>   |
| <code>assert_instance_of(class, object, message)</code>        | <p>Passes if the given object is an instance of the given class.</p> <p>For example:</p> <pre>assert_instance_of String, "IronRuby"</pre>   |
| <code>assert_kind_of(class, object, message)</code>            | <p>Passes if the given object is an instance of the given class or if class is one of its superclasses or one of the modules included in the object.</p> <p>For example:</p> <pre>assert_kind_of Numeric, 5.5</pre> |
| <code>assert_respond_to(object, method, message)</code>        | <p>Passes if the given object contains a definition of the given method.</p> <p>For example:</p> <pre>assert_response_to "Shay", :downcase</pre>  |



TABLE 17.1 Assert Methods

| Method  | Description  |
|---|--|
| <code>assert_match(pattern, string, message)</code>     | <p>Passes if the string matches the regex pattern.</p> <p>For example:</p> <pre>assert_match /d/, "5"</pre>  |
| <code>assert_no_match(pattern, string, message)</code>  | <p>Passes if the string doesn't match the regex pattern.</p> <p>For example:</p> <pre>assert_no_match /d/, "five"</pre>  |
| <code>assert_nil(object, message)</code>                | <p>Passes if the object is <code>nil</code>.</p> <p>For example:</p> <pre>val = nil assert_nil(val)</pre>  |
| <code>assert_not_nil(object, message)</code>            | <p>Passes if the object is not <code>nil</code>.</p> <p>For example:</p> <pre>assert_not_nil "IronRuby"</pre>  |
| <code>assert_same(expected, actual, message)</code>     | <p>Passes if <code>actual.equal? expected</code> returns <code>true</code>.</p> <p>For example:</p> <pre>assert_same 1, 1</pre>  |
| <code>assert_not_same(expected, actual, message)</code> | <p>Passes if <code>actual.equal? expected</code> returns <code>false</code>.</p> <p>For example:</p> <pre>assert_not_same 5, 6</pre>   |
| <code>assert_raise(*args)</code>                        | <p>Passes if the given block raises one of the given exceptions. The <code>*args</code> list should consist of exception classes. The message, if it exists, should be passed as the last parameter.</p> <p>For example:</p> <pre>assert_raise(ArgumentError) do   raise ArgumentError end</pre> |

TABLE 17.1 Assert Methods

| Method  | Description  |
|---|--|
| <code>assert_nothing_raised(*args) {}</code>                      | <p>Passes if the block doesn't raise an exception.</p> <p>The <code>*args</code> is there to specify expected exceptions and the message. If an exception class is specified, the test fails when it is raised. For other exception types, the test fails with the error. Default is any exception.</p> <p>If you want to pass a message, it must be passed as the last argument.</p> <p>For example:</p> <pre>assert_nothing_raised do   1 + 1 end assert_nothing_raised(ArgumentError) {   # This will cause the test to fail but   # without throwing an error.   raise ArgumentError }</pre> |
| <code>assert_throws(expected_symbol, message, &amp;proc)</code>   | <p>Passes if the block throws the expected symbol.</p> <p>For example:</p> <pre>assert_throws :success do   throw :success end</pre>   |
| <code>assert_nothing_thrown(message, &amp;proc)</code>            | <p>Passes if the block doesn't throw anything (an Exception instance or any other objects).</p> <p>For example:</p> <pre>assert_nothing_thrown do   1 + 1 end</pre>  |
| <code>assert_operator(object1, operator, object2, message)</code> | <p>Passes if <code>object1 operator object2</code> return true.</p> <p>For example:</p> <pre>assert_operator 1, ==, 1</pre>  |
| <code>assert_send(send_array, message)</code>                     | <p><code>send_array</code> consists of three items (an object, a method to execute, and the arguments to pass to the method).</p> <p>The assert passes if invoking the method on the object with the given arguments returns true.</p>   |
| <code>flunk(message)</code>                                       | <p>Always fails.</p> <p>For example:</p> <pre>flunk "test not implemented yet"</pre>   |

Notice that every assert method has an optional message argument. This message will be printed when the test fails (in addition to the regular failure message).

## Setup and Teardown

Every test case class contains two methods: setup and teardown. These methods are blank by default, but they can be overridden by the test case class. The setup method is run before every test method is executed, and the teardown method is invoked after the test method is done.

The setup method is the place to write code that is needed by all the tests (or most of them, at least). For example, you can initiate the tested object in the setup method and save it to an instance variable so that the test method can use it.

### GOAL OF OBJECTS INITIATED IN THE SETUP METHOD

Each test should focus on a very single method and therefore tests should not share state. This affects the objects you initiate in the setup method—the objects should not be used to share state between test methods. If they carry state, they should be released on the teardown method.

The teardown method is the place to write code that frees resources taken by the test. For example, if you open a connection to a database, create a heavy UI object, or open a file, the teardown method is where you should close or delete these objects. The teardown method is run even if the test fails or raises an unexpected exception.

In the preceding sample we have two methods, and they both create an instance of the CLR object. We can do that on the setup method and leave the actual tests to the methods:

```
require 'test/unit'
require "NumerologicalCalculator.dll"
class TC_NumerologicalCalculatorPublicTest < Test::Unit::TestCase
  def setup
    @instance = Numerology::Calculator.new
  end

  def teardown
    # Close connections, delete files, etc.
  end

  def test_string
    result = @instance.get_string_numerological_value "Shay"
    assert_equal 8, result
  end
end
```

```

def test_single_character
  result = @instance.get_string_numerological_value "a"
  assert_equal 1, result
end
end

```

## Test Suites

As the number of test cases rises, it will become harder to keep track of all of them, and you might forget to run some. Test suites provide a way to gather several test cases into a single file and execute this file instead of running each test case file individually.

There are two ways to create a test suite: explicitly or implicitly.

### Creating Test Suites Explicitly

Every test case you create contains an implementation of a method named `suite`. This method returns all the test cases and test methods in the class. The test framework takes this list and executes everything inside it.

To gather various test cases together, all you need is to create a class with a `suite` method that joins multiple suites into a single one.

For example, assuming we have two test case classes, `TC_NumerologicalCalculatorPublicTest` and `TC_NumerologicalCalculatorPrivateTest`, this is how we combine them both into a single test suite explicitly:

```

class TS_NumerologicalCalculatorTests
  def self.suite
    # Create a test suite named "Numerological Calculator Tests"
    suite = Test::Unit::TestSuite.new("Numerological Calculator Tests")
    suite << TC_NumerologicalCalculatorPublicTest.suite
    suite << TC_NumerologicalCalculatorPrivateTest.suite
    return suite
  end
end

```

### Creating Test Suites Implicitly

The test framework tries to make it easier for you to use it. Every file that requires the test/unit framework is automatically searched for tests when it is run. Therefore, all you have to do to run multiple test cases is to have them all in the same file.

It doesn't mean that you must write one huge file with tests. To gather some files together, just require them on the test suite file.

For example, assuming `TC_NumerologicalCalculatorPublicTest` is found within `public_tests.rb` and `TC_NumerologicalCalculatorPrivateTest` is found within `private_tests.rb`, the following sample code is equivalent to the explicit test suite sample:

```
require 'test/unit'
require 'public_tests.rb'
require 'private_tests.rb'
```

## NESTED TEST SUITES

An important capability of test suites is that they can also contain test suites. You can nest test suites and test cases in each other in the way you find most appropriate for the job at hand.

For example, you can create a test tree where the top test suite contains all tests of the application and the smaller suites contain tests for smaller portions of the application.

## Running the Tests

To this point, we've examined how to create tests and gather them. However, we still haven't seen how to actually run them.

Running a test file is as simple as it can be: Just execute the file with the test case classes like any other IronRuby file. When you require the test framework (test/unit), it automatically knows to execute the tests on the file when the file is executed.

So to run our public\_tests.rb file, open the command prompt and run the following:

```
ir public_tests.rb
```

This will run all test suites and cases on the file or the files required within it.

### Running a Specific Test Case or Suite

If you don't want to run the whole file but only a selected test case or suite, you have two options for doing so.

The first one, which is capable of running a specific test case but not a specific test suite, is to provide a command-line argument `-t` or `--testcase=`, specifying the name of the test case:

```
ir all_tests.rb -t TC_NumerologicalCalculatorPublicTest
```

Or

```
ir all_tests.rb --testcase=TC_NumerologicalCalculatorPublicTest
```

This runs the public test class only, even if there are other suites or cases in the file.

Patterns also can be passed as the `--testcase` value. For example, if you want run all test cases on the files, you can use a regex pattern:

```
ir all_tests.rb -t /TC_\s*/
```

The second way to run a specific test is via code. We can supply the code to run the test instead of the automatic one provided by the unit test framework. When the framework indicates that the code already executes tests, it suppresses the automatic runner.

To run a test case or suite via code, we use the console runner. The following code executes the test suite `TS_NumerologicalCalculatorTests`:

```
require 'test/unit/ui/console/testrunner'
Test::Unit::UI::Console::TestRunner.run TS_NumerologicalCalculatorTests
```

### Running a Specific Test Method

Running a specific test method is available through the command-line argument `-n` or `--name=`. Like the `--testcase` argument, this one can receive a full name or a regex pattern.

For example, if we want to run the test `test_string` only, we run the following statement from the command prompt:

```
ir public_tests.rb -n test_string
```

Or

```
ir public_tests.rb -name=test_string
```

This argument and the `--testcase` argument can also be combined to specify a test method (or methods by using a pattern) in a particular test case.

### Running Tests of Private Methods

As mentioned previously, there is no problem with testing private CLR methods. The only thing to remember is that you should use the `-X:PrivateBinding` switch:

```
ir -X:PrivateBinding private_tests.rb
```

Or use command-line arguments, as follows:

```
ir -X:PrivateBinding private_tests.rb --testcase=/TC_\s*/
```

## RSpec

RSpec is a unit test framework that works by the principles of behavior-driven development (BDD). It provides a special DSL for describing the expected behavior of the application. A *behavior* in the RSpec framework refers to a test container (like test case in the `Test::Unit` framework), and an *example* refers to the test method. The logic behind the terms is to make you think more about a behavior than structure; you have a behavior that contains examples of how it is expected to behave. You allegedly do not test code; you validate behavior.

For detailed documentation about RSpec, visit the project web page at <http://rspec.info> and the documentation part at <http://rspec.info/documentation>.

## Install RSpec

Before we start to actually use RSpec, we need to install it. RSpec comes as a Ruby Gem, so we use the `igem` tool to install it (which is located in the IronRuby installation folder).

Open the command prompt and execute the following command:

```
igem install rspec
```

When the command is done, you are ready to start using RSpec.

Although this is enough and lets you run RSpec tests, it is recommended to create a simple batch file that wraps a built-in RSpec command-line tool and helps you in executing the tests. Follow the next steps to create the file:

1. Go to your IronRuby directory and create a file named `ispec.bat`.
2. Open the file for edit and insert the following content into it:

```
@ECHO OFF
```

```
@ir.exe "c:\ironruby\lib\ironruby\gems\1.8\bin\spec" %*
```

If you have installed IronRuby in a different location, replace `c:\ironruby` in the command above with the location of the IronRuby folder. If you use the IronRuby sources, the path will be `Merlin\Main\Languages\lib\ironruby\gems\1.8\bin`.

3. Save the file. Now if your IronRuby directory is on the Windows PATH environment variable, you can use RSpec from any directory via the `ispec` file you have just created.

Go ahead and test it. Open the command prompt and enter the following:

```
ispec --help
```

You should receive brief help content about the RSpec command-line arguments.

## Requiring Needed Libraries

To work with RSpec, you need to require its libraries in the test files.

There are two required ones, `rubygems` and `spec`, and these provide the main functionality for RSpec:

```
require "rubygems"
require "spec"
```

To test our Numerological calculator C# application, we need to require the CLR assembly, too:

```
require "NumerologicalCalculator.dll"
```

## Running Tests

If you have only a single test file, it is possible to require spec/autorun and execute the file using the ir executable.

For example, assuming we have a file named calculator\_spec.rb, we can add the next line to the file:

```
require "spec/autorun"
```

Afterward, open the command prompt and navigate to the directory where the file exists. Then just run the file like every other IronRuby file:

```
ir calculator_spec.rb
```

This approach is good for one or two test files, but it won't suffice for larger projects. This is why we created the ispec file, which can help us run multiple test files and even entire directories.

The ispec command can receive a single file to execute:

```
ispec calculator_spec.rb
```

Or a directory path to execute all spec files found within:

```
ispec specs
```

When a directory path is passed, it is possible to pass a pattern that limits the files that are executed. For example, the next command executes only files that start with numerology\_ and end with .rb:

```
ispec specs --pattern "**/numerology_*.rb"
```

It is also feasible to run specific examples by passing the --example argument. The following command executes examples named "calculator test":

```
ispec specs --example "calculator test"
```

For all the available command-line arguments, run the help command:

```
ispec --help
```



## Creating a Behavior with describe

Examples in RSpec are held in an object named a `Behavior`. This object is created via the `describe` method. The `describe` method must be accompanied by a description or a class name (or both).

In our case, we strictly test the `Numerology::Calculator` class, so we will pass it to the `describe` method:

```
describe Numerology::Calculator do
  # examples will come here
end
```

If we were testing integration between several classes, it would have been odd to pass only one of the classes to the `describe` method. This is why you can pass a string instead that describes the behavior:

```
describe "integration test between several assemblies" do
  # examples will come here
end
```

It is also possible to combine the class name and description, to provide as much data as possible to the one who's reading the code or running the tests:

```
describe Numerology::Calculator, "is tested for perfection" do
  # examples will come here
end
```

If we run the file with the preceding code sample inside and we tell ispec to output the strings, too (by using `--format s`), the output we receive will be clear to read and understand:

```
> ispec calculator_spec.rb --format s
Numerology::Calculator is tested for perfection
- should calculate the right value for 'Shay'
- should work for a single character
```

```
Finished in 0.2960169 seconds
```

```
2 examples, 0 failures
```

The lines underneath the behavior title (start with *should*) are the examples that are defined within the code that is passed to the `describe` method. In the next section, we add them to the code.

## Creating Examples with it

As mentioned previously, a behavior in the RSpec world contains examples that validate that it behaves as expected. If a behavior doesn't have examples, it is useless and actually will not be executed by RSpec at all.

The `it` method is used to create an example. You probably wonder what the purpose of this name is; the method receives a description as its first parameter, so using it as the method name gives the sample code a more human language feeling.

Let's add two sample methods that will test our `get_string_numerological_value` method. Note how readable and clear the code is:

```
describe Numerology::Calculator, "is tested for perfection" do
  it "should calculate the right value for 'Shay'" do
    instance = Numerology::Calculator.new
    result = instance.get_string_numerological_value "Shay"
    result.should == 8
  end

  it "should work for a single character" do
    instance = Numerology::Calculator.new
    result = instance.get_string_numerological_value "a"
    result.should == 1
  end
end
```

The most important line in every example is the line where the actual result is compared to the expected one. Without this line, the example won't really test anything.

## Expectation Methods

When you require the RSpec framework file, every object in the system is extended with two new methods: `should` and `should_not`. With these methods, you can verify whether the actual result of the example code is the expected one.

These two methods might be a bit underestimated at first glance, but they contain everything you need and even more.

Their strength is in what they receive as a parameter, which is called an *expression matcher*. A matcher is a class that contains methods that know how to compare two objects in a specific way. Several different matchers come out of the box with RSpec, and you can create one yourself if the existing ones do not satisfy your needs.

Table 17.2 describes all the out-of-the-box matchers available to you on RSpec.

TABLE 17.2 RSpec Expression Matchers

| Method                         | Description  |
|--------------------------------|--|
| <code>be_true</code>           | Passes if the tested object is true, false, or nil, accordingly.   |
| <code>be_false</code>          | For example:   |
| <code>be_nil</code>            | <code>(1 == 1).should be_true</code>   |
| <code>be_[predicate]</code>    | Passes if the tested object equals a given arbitrary predicate.<br>For example:<br><code>(1+2).should be_3</code>  |
| <code>be_a</code>              | Passes if the tested object is an instance of the given class or if class is   |
| <code>be_an</code>             | one of its superclasses or one of the modules included in the object.  |
| <code>be_kind_of</code>        | For example:<br><code>1.should be_a(Numeric)</code><br><code>[1,2].should be_an(Array)</code>  |
| <code>be_instance_of</code>    | Passes if the tested object is an instance of the given class.   |
| <code>be_an_instance_of</code> | For example:<br><code>"Hi".should_not be_instance_of(Numeric)</code>   |
| <code>be_close</code>          | Passes if the tested object equals or close to the value within delta tolerance.<br>For example:<br><code>1.5.should be_close(2, 0.5)</code>   |
| <code>change</code>            | Passes if the tested object value (or one of its attributes) is changed as expected after running a given proc.<br>To set the expected results, the following methods are available: <code>by</code> , <code>by_at_least</code> , <code>by_at_most</code> , <code>from</code> and <code>to</code> .<br>For example:<br><pre>arr = [] lambda {   arr &lt;&lt; 1 }.should change(arr, :size).by(1)  str = "Ruby" lambda {   str = "Iron" + str }.should change { str }.from("Ruby").to("IronRuby")</pre> |
| <code>Eq1</code>               | Passes if the tested object is the same as the expected object but not necessarily the same object.<br>For example:<br><pre>obj1 = 1 obj2 = 1 obj1.should eq1(obj2)</pre>  |

TABLE 17.2 RSpec Expression Matchers

| Method                                | Description  |
|---------------------------------------|--|
| Equal                                 | <p>Passes if the tested object and the expected object are the same object.</p> <p>For example:</p> <pre>1.should equal(1) # Fixnums are the same object</pre>   |
| Have<br>have_at_least<br>have_at_most | <p>Passes if the tested object contains the expected number of items.</p> <p>Note that the have method should be preceded by the name of the collection. If the object itself is the collection, you can write whatever you like as the collection name.. (Something logical is recommended; for instance, items or members.)</p> <p>For example:</p> <pre>[1, 2].should have(2).items</pre>   |
| include                               | <p>Passes if the tested object includes the value.</p> <p>For example:</p> <pre>[1,2,3].should include(1) [1,2,3].should include(1,2) "IronRuby".should include("Ruby")</pre>  |
| Match                                 | <p>Passes if the tested object matches the given regular expression pattern.</p> <p>For example:</p> <pre>"IronRuby".should match(/Iron[\s]*/)</pre>   |
| raise_error                           | <p>Passes if the tested object raises an error.</p> <p>The different variations of raise_error are presented on the following samples:</p> <pre>err = lambda {   raise StandardError, "Oh no!" }</pre> <pre># Passes on every raised error err.should raise_error # Passes when a StandardError is raised err.should raise_error(StandardError) # Passes when a StandardError is raised with # the message "Oh no!" err.should raise_error(StandardError, "Oh no!") # Passes when a StandardError is raised with a message # that matches the given pattern err.should raise_error(StandardError, /no!/) # Passes when an error is raised and the message length # is shorter than 10 characters err.should raise_error do  error    error.message.length.should &lt; 10 end</pre> |

TABLE 17.2 RSpec Expression Matchers

| Method                    | Description  |
|---------------------------|--|
| <code>respond_to</code>   | <p>Passes if the tested object responds to all of the names or symbols provided.</p> <p>For example:</p> <pre>"Hi".should respond_to :reverse, "upcase"</pre>                      |
| <code>satisfy</code>      | <p>Passes if the given block returns true. The tested object is passed to the block.</p> <p>For example:</p> <pre>"IronRuby".should satisfy do  obj    obj == "IronRuby" end</pre> |
| <code>throw_symbol</code> | <p>Passes if the tested proc throws the expected symbol.</p> <p>For example:</p> <pre>lambda { throw :good }.should throw_symbol(:good)</pre>                                      |

## Before and After

The RSpec framework allows you to inject code in several places. This is good for logging purposes, initialization, disposing of heavy objects, and so on.

### Before and After Each Example

To run code before and after each example is executed, the `before(:each)` and `after(:each)` methods can be used.

For example, the following code initializes our `Calculator` class before every example and saves the instance to an instance variable:

```
describe Numerology::Calculator, "is tested for perfection" do
  before(:each) do
    @instance = Numerology::Calculator.new
  end
end
```

### Before and After a Behavior

It is possible to inject code before the behavior examples are invoked and after they are finished. This is done with the same `before` and `after` methods from before, just with a different parameter value passed to them: `before(:all)` and `after(:all)`.

The next sample prints a message to the screen before and after the behavior is executed:

```
describe Numerology::Calculator, "is tested for perfection" do
  before(:all) do
    puts "Starting behavior"
  end
end
```

```

after(:all) do
  puts "Finished behavior"
end
end

```

### Global Before and After

It is also possible to assign code to global before and after methods. This code will be the first and last to run.

These global hooks should be defined on a file that all spec files require. In this file, you should add the global before and after code as follows:

```

Spec::Runner.configure do |config|
  # Will run before every behavior
  config.before(:all) {}
  # Will run before each example
  config.before(:each) {}
  # Will run after each behavior
  config.after(:all) {}
  # Will run after each example
  config.after(:each) {}
end

```

### Order of Execution

There are multiple possibilities to add code before and after different operations. As a result, it might become unclear when the code is actually executed. Figure 17.1 attempts to make that clear and lays out the order of their execution.

## Cucumber

Cucumber is another test framework that matches the BDD agenda, and it's the most interesting one.

The Cucumber framework lets you define behavior rules in a unique DSL called *Gherkin*, which is as close as a DSL can be to regular English (or one of the other available language packs for Cucumber).

A unit test file in Cucumber consists of a feature declaration, scenarios, and scenario steps. The feature contains scenarios and can be declared with a plain English description. Every feature scenario has a name, and it contains a list of steps that are translated afterward to code.

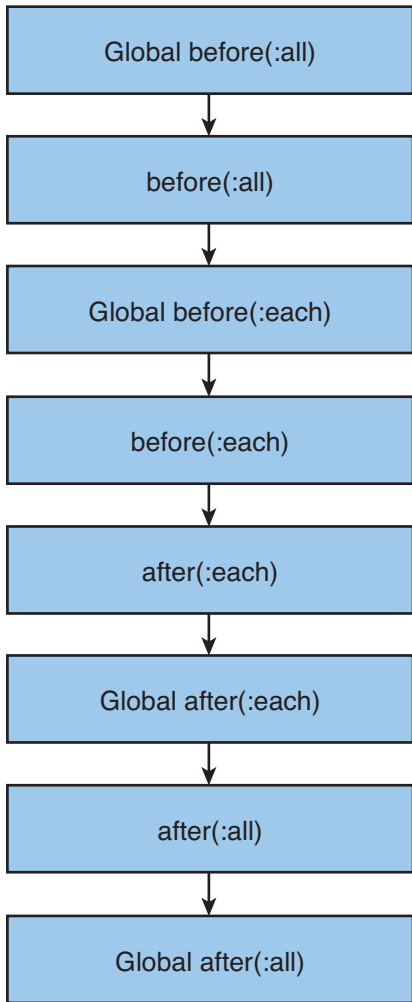


FIGURE 17.1 Order of execution of before and after methods.

For example, the following code is a valid Cucumber test:

Feature: Make a Salad

If we want to stay healthy  
we must eat salad. The application  
should cut vegetables successfully every time.

Scenario: Cut Vegetables

Given 1 tomato for the salad

```
And 3 cucumbers for the salad
When I press the cut button
Then salad should be ready
```

You can access the official Cucumber website at <http://cukes.info>, where you can find detailed documentation and various examples.

## Installing Cucumber

Just like RSpec, Cucumber is a Ruby Gem. To install it, open the command prompt and execute the following command:

```
gem install cucumber
```

The command installs several gems that Cucumber also uses.

Now we create a wrapper script file that makes using Cucumber easier. We call it `icucumber`. The file shortens the command we need to write to use Cucumber. Follow the next steps to create the file:

1. Go to your IronRuby directory and create a file named `icucumber.bat`.
2. Open the file for edit and insert the following content into it:

```
@ECHO OFF
@ir.exe "c:\ironruby\lib\ironruby\gems\1.8\bin\cucumber" %*
```

If you have installed IronRuby, you should replace `c:\ironruby` in the preceding command with the location of the IronRuby folder. If you use the IronRuby sources, the path will be `Merlin\Main\Languages\lib\ironruby\gems\1.8\bin`.

3. Save the file. Now if you're IronRuby directory is on the Windows PATH environment variable, you can use Cucumber from any directory via the `icucumber` file you just created.

Go ahead and test it. Open the command prompt and enter the following:

```
icucumber --help
```

You should receive brief help content about the Cucumber command-line arguments.

## Project Structure

Cucumber project files should be separated into different logical directories under a root folder named `features`. The full folder structure is shown in the following table.



| Folder Name               | Description   |
|---------------------------|---|
| features                  | Root directory. Holds all feature files.  |
| features/step_definitions | Contains the files that define the step definitions.                                  |
| features/support          | Contains files with support code for the features. Contains the env.rb file, as well. |

In addition, using multiple folders can help to separate logical parts of the application. For example, a sample project structure might be as follows:

```
features
-- Server
  -- step_definitions
-- Client
  -- step_definitions
  -- step_definitions
  -- support
```

Another advantage of using multiple directories is that it is possible to execute the features of a specific folder only. On the preceding sample folder structure, for instance, it is possible to execute only server related tests.

### The env.rb File

The env.rb file is a file where you can put Cucumber environment related code. For example, hooks can go there or special initialization code.

The file should be placed within the support folder. It is an optional file, and you can use any other filename for it, as long as the file is located in the support folder.

## Features

A feature in Cucumber represents a feature of the application. It contains a set of scenarios that are expected to make sure the tested feature works.

A feature file should have the extension of .feature. It also has a format that should be followed:

```
Feature: <title>
  <description>
  <scenario 1>
  <scenario 2>
  ...
```

The title and description are free text arguments. The description is also multiline and is intended to explain what the feature is about, how it is expected to work, and every other detail you find important to document.

### WHAT TO INCLUDE IN THE FEATURE DESCRIPTION

The feature description is there so that everyone who reads the feature file understands what it is about and why. It is recommended to use a format for the description so that all the needed information will be included.

The format shouldn't be strictly enforced, but it's recommended to stick to one. The recommended details to include in the description are as follows:

- ▶ The user who is expected to use the feature (for example, “a regular user” or “a power user”)
- ▶ What the feature does (for example, “sends an alert email”)
- ▶ The reason why the feature is needed (for instance, “to warn everybody and prevent data loss”)

To test our .NET code, we start by creating a feature named `calc_numerological_value.feature`. The feature definition follows:

```
Feature: Numerological Value Calculation
  As a regular creature living on earth
  I want to know my numerological number
  To know my future
```

This file should be saved under the features folder. If you have defined a folder tree to logically separate the features, save this file in the correct subfolder. For example, in the sample structure on the previous section, we could put this file under the Client folder instead of the root folder.

The next section describes our next task—to define the actual test scenarios.

## Scenarios

A scenario is a part of a feature. It consists of a sequence of steps that form a single test.

The scenario format is as follows:

```
Scenario: <title>
  <step 1>
  <step 2>
  ...
```

A scenario is built from three types of steps:

- ▶ **Given:** Sets up preconditions for the scenario (for example, the input to the feature, the context)

- ▶ **When:** The behavior that is being tested (for instance, clicking a button, executing an operation)
- ▶ **Then:** Defines the expected result (for example, “the output should be 8” or “‘Hello’ should be printed on the screen”)

Another step directive, which doesn’t belong to a specific type, is **And**, which provides a way to add another statement of the last step type. It can be used with any step type. For example, consider the next scenario:

```
Scenario: ask the rabbit
  Given I love carrots
  Given I love lettuce
  When I am asked what I am
  Then I should answer "rabbit"
  Then I should run away
```

The **And** directive makes this scenario easier for humans to read. Consider how the preceding scenario looks with **And** (the scenarios are equal):

```
Scenario: ask the rabbit
  Given I love carrots
  And I love lettuce
  When I am asked what I am
  Then I should answer "rabbit"
  And I should run away
```

For our little application, we need a scenario to test the numerological calculation result. The next scenario checks that:

```
Scenario: Calculate the value of Shay
  Given I have inserted "Shay" to the calculator
  When I calculate
  Then the result should be 8
```

This scenario wraps up the feature file and eventually, the `calc_numerological_value.feature` looks as follows:

```
Feature: Numerological Value Calculation
  As a regular creature living on earth
  I want to know my numerological number
  To know my future
```

```
Scenario: Calculate the value of Shay
  Given I have inserted "Shay" to the calculator
  When I calculate
  Then the result should be 8
```

## Implementing Steps

The step implementation is defined in an RB file, which can be placed on the same directory of the feature or on the `step_definitions` folder (which is the recommended place).

A steps definition file includes the code that interprets the `Given`, `When`, and `Then` directives. There is no need for a class or a module; the code is placed in the file directly on the global context.

The format of the different directives is similar. The generic one is as follows:

```
<Given or When or Then> <regex pattern> do
  ... code ...
end
```

As you can see, Cucumber uses regular expressions to interpret the textual step commands. A simple string is also a possibility, but regex gives much better control over the input and is recommended over plain strings.

The strength of this way of interpreting strings is the option to extract data from it. This is done using regular expression groups. For example, when given a string "When you wish upon a star" and the pattern `/When you (.*) upon a (.*)/`, you can use `wish` and `star` as variables in the code afterward.

To pass variables between steps, just use instance variables.

Okay, so now that you know how to implement the steps in general, let's create a step definition file for our numerology calculator feature we wrote earlier.

The first task for us is to require the CLR DLL file. This way we can use it within our steps:

```
require "NumerologicalCalculator.dll"
```

With the tested assembly ready for us, we can move on to the step definitions.

From the `Given` part of our scenario, "Given I have inserted "Shay" to the calculator", you can see that we need to extract what was inserted to the calculator. We can extract this data item and save it to the next steps:

```
Given /I have inserted "(.*)" to the calculator/ do |name|
  @name = name
end
```

The extracted groups are passed to the associated code block via the block arguments. If we had, for example, two groups within our pattern, then we would have two block arguments, as well.

Now to the next step: `When`. The `When` step is the one where the action is executed. In our application, this step doesn't provide any additional data, it just points out that all the preconditions have been specified (by the `Given` steps) and that the action should be executed.

Hence, we execute the calculation on this step:

```
When "I calculate" do
  instance = Numerology::Calculator.new
  @result = instance.get_string_numerological_value @name
end
```

Notice that we used here the @name variable that we had saved on the Given step.

The last step is the Then step. Here we need to validate that the result is the expected one. We extract the expected result from the Then string and compare it to the result from the calculator (which we have received on the When step):

```
Then /the result should be (.*)/ do |result|
  @result.should == result.to_i
end
```

## COMPARING THE ACTUAL AND EXPECTED RESULTS

Note the way we validate the result: We use the should method for that. This should method comes from the RSpec framework, which Cucumber takes advantage of. All should and should\_not variations are available for you to use.

Another issue to note is that the values extracted from the string (via the regular expression) are always strings. You have to convert them to the expected type before comparing to the actual result.

Save the file, and we're ready to run the test. Open the command prompt and navigate to the folder containing the features directory. For example, if the features directory is found at C:\Projects\NumerologicalCalc\Features, navigate to C:\Projects\NumerologicalCalc. When you're in the directory, run the next command:

```
icucumber features --no-color --no-source
```

The test results display a few moments later:

```
Feature: Numerological Value Calculation
  As a regular creature living on earth
  I want to know my numerological number
  To know my future

Scenario: Calculate the value of Shay
  Given I have inserted "Shay" to the calculator
  When I calculate
  Then the result should be 8
```

```
1 scenario (1 passed)
3 steps (3 passed)
0m0.294s
```

Additional information about running Cucumber tests appears later in this chapter.

### Scenario Outlines and Example Tables

It is pretty common to have several scenarios that differ only by the values they pass to the test framework. For example, if we want to test our calculator with different strings, we have to write the same scenario with only the string value and result different between the scenarios.

This is exactly where scenario outlines and example tables come to our aid. A scenario outline lets us describe a scenario with placeholders, and the example table is used to provide the values for the placeholders. This way you write the scenario once and just pass different value to it every time.

There are two differences between a regular scenario and a scenario outline:

- ▶ Scenario outlines begin with `Scenario Outline:` rather than `Scenario:` in regular scenarios.
- ▶ Scenario outlines contain placeholders that are defined between smaller-than and greater-than signs. For example, on the line `Given I have <amount> apples`, there is a single placeholder named `amount`.

The example table follows the scenario outline and supplies values for the placeholders. The example table is laid out as a table. The first line is the header line where each placeholder should have a column with its name. The next lines contain the values themselves; each line describes the values for a single scenario. The table columns are delimited by a vertical bar.

Let's convert our scenario from previous sections to a scenario outline with an example table. The following Gherkin code defines three scenarios using the scenario outline and example table technique:

```
Scenario Outline: Calculate values
  Given I have inserted "<name>" to the calculator
  When I calculate
  Then the result should be <result>
```

Examples:

|           |        |  |
|-----------|--------|--|
| name      | result |  |
| Shay      | 8      |  |
| IronRuby  | 5      |  |
| Unleashed | 8      |  |

When we run this feature, we see that three scenarios have been executed:

```
> icucumber features --no-color --no-source
Feature: Numerological Value Calculation
  As a regular creature living on earth
  I want to know my numerological number
  To know my future

Scenario Outline: Calculate values
  Given I have inserted "<name>" to the calculator
  When I calculate
  Then the result should be <result>

Examples:
  | name      | result |
  | Shay      | 8      |
  | IronRuby  | 5      |
  | Unleashed | 8      |

3 scenarios (3 passed)
9 steps (9 passed)
0m0.444s
```

## A Background

Sometimes you have the same preconditions on all features (when working as a specific user, setting some values, and so on). The Background directive enables you to write Gherkin code that runs before any scenario on the feature file.

For example, in the following feature code, I use the Background directive to define the user to work as in the feature scenarios:

```
Feature: give permissions
  As a system administrator
  I want to change permissions of files
  To enable other users to use them

Background:
  Given I work as user "sysadmin"
Scenario: change permissions to a restricted file
  Given I use "restricted_file"
  When I change its permissions to allow "Everyone" to write
  Then I should receive a warning
```

## Tags

By default, Cucumber executes all feature files and scenarios within the folder it's pointed to. However, sometimes you may want to test specific features or scenarios only (for example, when you want to run only the most basic tests). One way is to separate the features into different folders, but with this approach it will be quite complicated to run only the basic scenarios of each feature.

Tags make it possible to mark features or scenarios with a unique name and execute only them.

### Tagging Features and Scenarios

To tag a feature or a scenario, the tag identifier prefixed by an at sign (@) should be placed on the line before the feature or scenario definition.

#### TAG IDENTIFIER IS NOT AN INSTANCE VARIABLE

The use of the at sign (@) for the tag identifiers might be confused with Ruby's instance variables. However, it is important to notice that there is no connection between them because the tag identifiers are placed on the feature files that contain Gherkin code and not Ruby.

Several tags can be applied to a single feature or scenario by writing them one after the other, delimited by a space.

For example, in the following code sample, I tag the feature with @numerology and @calc\_numerological\_value tags; the first scenario is tagged with the @important tag:

```
@numerology @calc_numerological_value
Feature: Numerological Value Calculation

  @important
  Scenario: Calculate the value of Shay
    Given I have inserted "Shay" to the calculator
    When I calculate
    Then the result should be 8

  Scenario: Calculate the value of IronRuby
    Given I have inserted "IronRuby" to the calculator
    When I calculate
    Then the result should be 5
```



**Running Tagged Features and Scenarios**

To run only features and scenarios that are marked with a certain tag, the `--tags` argument should be used when executing the `icucumber` batch file.

It is possible to execute multiple tags by passing them separated by a comma.

For example, the next command runs all scenarios and features tagged with `@numerology` and `@important` tags:

```
icucumber features --tags @numerology, @important
```

The `--tags` argument can also receive the tags that should not be run. To do that, just add a tilde (`~`) before the tag name. The following command runs all features and scenarios except the ones marked with the `@important` tag:

```
icucumber features --tags ~@important
```

**Hooks**

The Cucumber framework enables you to add code that runs during the execution of the tests by using hooks. Hooks can be written in the `env.rb` file or in any other Ruby file that is placed in the `features/support` directory.

**Global Hooks**

Global hooks are executed when Cucumber starts and before it exits. To run code when Cucumber starts, just add it to the global context.

For example, to write print a message on the screen when Cucumber starts, just add the next line to the file where you host the hooks in:

```
puts "IronCucumber is starting"
```

The other side of the global hooks is the code that runs when Cucumber exits. This is done via the `at_exit` method, which receives a block of code. The following code prints a message to the screen when Cucumber exits:

```
at_exit do
  puts "IronCucumber ends"
end
```

**Scenario Hooks**

Scenario hooks execute code before and after every scenario is invoked.

The `Before` method is used to set the code to run before every scenario. The following code prints a message before every scenario:

```
Before do
  puts "A scenario is starting"
end
```

The `After` method is used to set the code to run after every scenario is done. The method passes to its associated code block an object that contains information about the scenario. The available methods are `failed?`, `passed?`, and `exception`.

The next code adds code to print an informative message after every scenario ends:

```
After do |scenario|
  if scenario.passed?
    puts "Scenario passed"
  else
    puts "Scenario failed: #{scenario.exception.message}"
  end
end
```

### Step Hooks

Cucumber also enables you to run code after each step is executed. This is done with the `AfterStep` method. The scenario object is passed to the step and lets you investigate it for failures.

The following code prints a message with the exception message whenever a step fails:

```
AfterStep do |scenario|
  if scenario.failed?
    puts "FAILURE: #{scenario.exception.message}"
  end
end
```

### Tagged Scenario Hooks

The last type of scenarios enables you to run code before and after scenarios with specific tags only.

This is done with methods you've seen previously on this chapter: `Before`, `After`, and `AfterStep`. To define which tags these methods are related to, just pass them the tag names as a comma-separated list.

For instance, the following code shows a message before a scenario tagged with `@high_priority` or `@critical` tags:

```
Before('@high_priority', '@critical') do
  puts "THIS IS AN IMPORTANT STEP"
end
```

## A World

When a Cucumber scenario is run, it runs inside a world. A world is actually an instance of an object. Because the scenarios run inside the world, we can use it to share helper methods, logging, and so forth.

To do that, you must create a class with whatever methods you want and then tell Cucumber to use it in its world.

For example, in the next sample, I create a class with a single method that writes messages to the log, and then I add it to the framework world. This code should be added to the `env.rb` file or any other Ruby file within the support folder:

```
def HelperMethods
  def add_log(message)
    # Write to log...
  end
end

# Add the class to the Cucumber world
World do
  HelperMethods.new
end
```

Now the `add_log` method is available to any step definition in the code.

## Multilanguage

Up to this point, I have shown you sample Cucumber code in English only. Cucumber doesn't ignore those who speak languages other than English. Cucumber does enable users to write feature specification in various languages (such as Spanish, German, Russian, Hebrew, and even funny ones like LOL code). To view all available languages, open the command prompt and execute the following command:

```
icucumber --language help --no-color
```

If you want to see the keyword translation, open the `languages.yml` file, which is located under the Cucumber directory at `%Cucumber Folder%/Lib/Cucumber/languages.yml`. A translation part, for Danish in the next sample, looks like this:

```
"da":
  name: Danish
  native: dansk
  encoding: UTF-8
  feature: Egenskab
  background: Baggrund
  scenario: Scenarie
  scenario_outline: Abstrakt Scenario
  examples: Eksempler
```

```

given: Givet
when: Når
then: Så
and: Og
but: Men
space_after_keyword: true

```

When you prepare to use a language other than English, make sure that the patterns on the code match the language.

To run Cucumber with a language other than English, add the `--language` argument to the `icucumber` command followed by the language name.

For example, the next command runs the tests using the Polish language:

```
icucumber features --language pl
```

## Executing Cucumber

The Cucumber framework has several command-line arguments and switches.

The simplest Cucumber command requires a relative path to the features directory:

```
icucumber features
```

Or with a longer path:

```
icucumber my_project/features
```

It is also possible to define a specific features file:

```
icucumber features/my_feature.feature
```

This goes even further: You can specify the line in the file where the scenario you want to execute exists. When the line number is passed, only the scenario on this line will be executed. In the next example, the scenario on line 12 in the `my_feature.feature` file is executed:

```
icucumber features/my_feature.feature:12
```

It is also possible to run scenarios with a specific name by using the `--name` argument. The following sample executes scenarios that are named "test a string":

```
icucumber features --name "test a string"
```

These are just a few of the available commands. To review all available command-line arguments and switches, use the `help` command:

```
icucumber --help
```

## Summary

In this chapter, you were introduced to three different test frameworks: `Test::Unit`, `RSpec`, and `Cucumber`. Each framework has its advantages and disadvantages, and you should use the one that you feel most comfortable with.

In this chapter, you learned the basics of the different test frameworks. These suffice in most cases, but I still highly recommend you to read further and delve deeper into these frameworks when you start working with them.

If you didn't like these frameworks, don't give up just yet. Several other test frameworks that you might like are available in the Ruby language. These include `Shoulda`, `test-spec`, `expectations`, `webrat` for web testing, and more. Just search the Internet for them, and you'll find a plethora of documentation and examples.

## CHAPTER 18

# Using IronRuby from C#/VB.NET

### IN THIS CHAPTER

- ▶ Hello, IronRuby from CLR
- ▶ The Classes of the Process
- ▶ Executing IronRuby code from C#/VB.NET

In this book so far, we have discussed in depth how to use CLR objects from IronRuby in various different ways. This chapter is different because it goes the other way around: You learn how to embed and run IronRuby code from your static languages code.

In this chapter, you are shown how to create a C#/VB.NET project in Visual studio that can run IronRuby code. After that, we delve into the DLR classes that take part in the process of running IronRuby code, and in the last part of this chapter you are introduced to some real-world scenarios and their solutions.

## Hello, IronRuby from CLR

Thanks to the DLR, calling IronRuby code from other languages is a matter of a few lines of codes.

In this section, you see how easy this task is, all while you create your first project that runs IronRuby from a static CLR language.

So let's start. The first task is to create a project in Visual Studio:

1. Open Visual Studio. The instructions here are targeted to Visual Studio 2008, but they will be the same or very similar in other versions of Visual Studio.
2. Go to File > New > Project.
3. On the left panel, choose Visual C#/Windows or Visual Basic/Windows.
4. On the right panel, choose Console Application.

5. We are going to work on a console application, but it is possible to use IronRuby from any project type.
6. In the lower panel, insert the project name **IronRubyViaCLR**, choose a location for the solution, and click OK.

Figure 18.1 shows what the dialog should look like before you click OK.

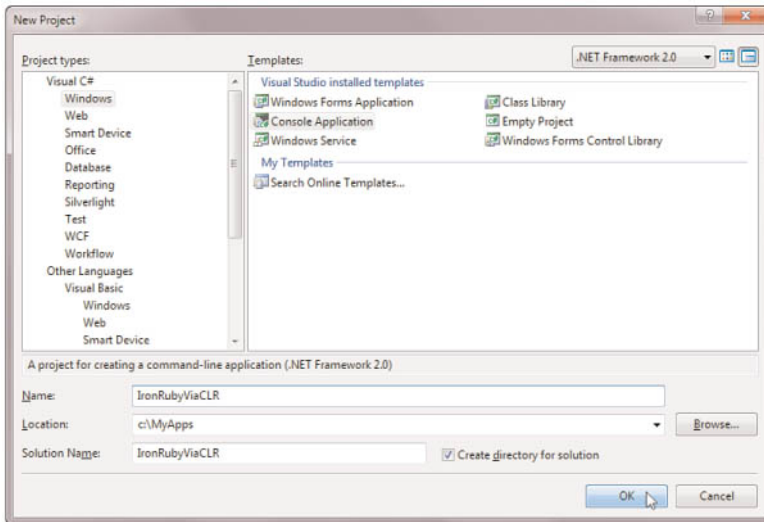


FIGURE 18.1 New Project dialog as it should be filled.

Now that we have our solution ready, we need to add references to IronRuby and DLR assemblies:

1. When the solution is opened in Visual Studio, go to Project > Add Reference.
2. In the Add Reference dialog, go to the Browse tab.
3. Navigate to your IronRuby folder.
4. Choose IronRuby.dll, IronRuby.Libraries.dll, Microsoft.Scripting.dll, Microsoft.Scripting.Core.dll (you can choose multiple DLLs by keeping the Ctrl key pressed), and click OK.

Currently, you have a Visual Studio project that is ready to work with IronRuby. Every time you want to embed IronRuby in a C# or a VB.NET project, redo these steps and continue with writing the actual code.

### IRONRUBY CAN BE EMBEDDED INTO EVERY PROJECT TYPE

Even though all the samples in this chapter use a console application project, it is not the only project type you can embed IronRuby into.

IronRuby can be embedded into every project type. Just add the needed assemblies to the referenced assemblies list and you're ready to go.

The next step is to write the actual code. The next code runs the IronRuby code puts 'Hello from IronRuby'.

In C#

```
using Microsoft.Scripting.Hosting;

class Program
{
    static void Main(string[] args)
    {
        ScriptEngine engine = IronRuby.Ruby.CreateEngine();
        ScriptSource source =
            engine.CreateScriptSourceFromString(
                "puts 'Hello from IronRuby'");
        source.Execute();
    }
}
```

In VB.NET

```
Imports Microsoft.Scripting.Hosting

Module Module1
    Sub Main()
        Dim engine As ScriptEngine = IronRuby.Ruby.CreateEngine()
        Dim source As ScriptSource = _
            engine.CreateScriptSourceFromString(
                "puts 'Hello from IronRuby'")
        source.Execute()
    End Sub
End Module
```

Running this code prints “Hello from IronRuby” onscreen.

Other than just running textual scripts, the DLR enables us to execute entire files, pass data between the languages, and even use IronRuby classes and methods in C# and VB.NET.

We start by getting to know the DLR classes involved.

## The Classes of the Process

The DLR contains different classes, each with a goal in the script execution life. Let’s go through the important ones: `ScriptRuntime`, `ScriptEngine`, `ScriptScope`, and `ScriptSource`.



## ScriptRuntime

The `ScriptRuntime` class is the top-level object. It contains the runtime configuration (which can be passed on initialization), the “global scope,” which is available to all the scripts running within this runtime, the referenced assemblies, and the available language engines. To better isolate running scripts, multiple runtimes can be created on a single app domain.

### Creating a ScriptRuntime

There are a few ways to create a script runtime.

The first one is to use the Ruby class static method `CreateRuntime`. This method does not require a configuration object because it uses IronRuby’s configuration by default:

```
ScriptRuntime runtime = IronRuby.Ruby.CreateRuntime();
```

The second way is by calling the `ScriptRuntime` constructor where you won’t be able to avoid passing the configuration object. Here we just pass an empty configuration (but we will talk more about `ScriptRuntimeSetup` shortly):

```
ScriptRuntimeSetup conf = New ScriptRuntimeSetup()
ScriptRuntime runtime = New ScriptRuntime(conf)
```

The third way to create a `ScriptRuntime` object makes it possible to create the runtime on a different app domain. This is done via the static method of the `ScriptRuntime` class named `CreateRemote`:

```
AppDomain domain = AppDomain.CreateDomain("IronRuby app domain");
ScriptRuntimeSetup conf = new ScriptRuntimeSetup();
ScriptRuntime s = ScriptRuntime.CreateRemote(domain, conf);
```

### ScriptRuntimeSetup

It is possible to provide startup configuration to the script runtime via the `ScriptRuntimeSetup` class.

The class contains a few properties that provide some control over the runtime execution. They are described in Table 18.1.

TABLE 18.1 `ScriptRuntimeSetup` Properties

| Property Name              | Description   |
|----------------------------|---|
| <code>DebugMode</code>     | Determines whether the runtime is run in debug mode.<br>When the runtime is run in debug mode, it runs without debug symbols and without any JIT optimizations. |
| <code>HostArguments</code> | Arguments that are passed to the host when it is constructed.   |

TABLE 18.1 ScriptRuntimeSetup Properties

| Property Name  | Description   |
|----------------|---|
| HostType       | The type of the host of the runtime. Intended to make it possible to run the runtime in different environments (such as Silverlight).<br>The type can be any class that inherits from the ScriptHost class. |
| LanguageSetups | A list of LanguageSetup instances that contain language-specific configuration like the language display name or file extensions.   |
| Options        | A dictionary of string-object pairs with miscellaneous configuration items.<br>For example, this is used to define directories for the search path in IronRuby.   |
| PrivateBinding | Determines whether CLR visibility should be ignored.  |

### Executing Code

The `ScriptRuntime` class contains a way to execute IronRuby code. However, it provides less control over the operation compared to other approaches.

Executing IronRuby code is done via the `ExecuteFile` method. It receives a path to a dynamic language code file, realizes the language (by file extension), executes it, and returns a `ScriptScope` object that describes the context in which the code was executed.

The next sample executes the IronRuby file that exists in `C:\test.rb`:

```
ScriptRuntime runtime = IronRuby.Ruby.CreateRuntime();
runtime.ExecuteFile("test.rb");
```

## ScriptEngine

A script engine is the main object that represents a DLR language. All our interaction with IronRuby starts from this class.

### Creating a ScriptEngine

There are two ways to create an IronRuby script engine: the direct one and the generic one.

The direct way is done via the `Ruby` class:

```
ScriptEngine engine = IronRuby.Ruby.CreateEngine();
```

The generic one allows creating script engines for other installed DLR languages (like IronPython):

```
Dim runtime As ScriptRuntime = IronRuby.Ruby.CreateRuntime()
Dim engine As ScriptEngine = runtime.GetEngineByFileExtension("rb")
```

**Search Paths**

When IronRuby code loads, it might require different files or libraries. In such case, IronRuby goes to the engine search paths and looks for the file there.

The `ScriptEngine` class provides two methods to access and alter the search paths: `GetSearchPaths` and `SetSearchPaths`. These methods return or set a collection of strings that represent the paths to look in.

The next sample adds "C:\libs" to the search paths of the engine:

```
ICollection<string> paths = engine.GetSearchPaths() ?? new List<string>();
paths.Add(@"\libs");
engine.SetSearchPaths(paths);
```

**Executing Code**

Script engine can also execute IronRuby code. It provides four methods for doing so, and each method can help you in a different situation.

`Execute` receives a string that represents the code to execute, runs it, and returns the result as an object. It can also run the code in a specific scope (as well as the other methods we will see), as discussed in the "ScriptScope" section, later in this chapter.

The next sample executes `1+1` with IronRuby, gets the result, and uses it:

```
object result = engine.Execute("1+1")
int num = Convert.ToInt32(result) + 8
' Prints 10 on the screen
Console.WriteLine(num)
```

`Execute<>` is identical to `Execute`, with a single difference: The return value will be converted to the type passed as the generics parameter. For example, the following code is doing exactly what the previous one did, just without saving the result as an `Object` and converting it to `Int32`:

```
int num = engine.Execute<Int32>("1+1");
num = num + 8;
Console.WriteLine(num);
```

`ExecuteFile` executes a file and returns a `ScriptScope` object representing the scope it was executed in. Just like the regular IronRuby interpreter, if you don't have code to run (for example, everything is inside modules, classes, and methods), nothing will be actually executed at this stage. You must have code in the global context to do that.

For example, the following IronRuby code exists within a file named `test.rb`, which is located under `C:\Libs`:

```
def add(a, b)
  a+ b
end
```

```
puts add(1,2)
puts add(12.5, 25.3)
puts add("Iron", "Ruby")
```

To run it, we can take advantage of `ExecuteFile`:

```
engine.ExecuteFile(@"Libs\test.rb")
```

This prints the following output onscreen:

```
3
37.8
IronRuby
```

`ExecuteAndWrap` is similar to `Execute`, only it returns a `System.Runtime.Remoting.ObjectHandle` object. This object can be passed between app domains, and it is recommended to use this method only for such cases.

## ScriptScope

The `ScriptScope` object represents a context. It doesn't have a lot of methods, and the ones that it does have are all variable related. You can set, get, and remove variables from the scope with a very easy API. These variables are then available to the code that runs with this scope.

### Creating a ScriptScope

You can create a `ScriptScope` either from the script runtime or from script engine instances.

Via the `ScriptRuntime` instance, you can use the `CreateScope` method. Here you can also pass a language name that defines the dynamic language this scope is related to:

```
// General scope, not language specific
runtime.CreateScope();
// IronRuby scope
runtime.CreateScope("IronRuby");
```

Through the `ScriptEngine` instance, there are two ways of creating a scope:

```
// Create a scope from the file content
engine.ExecuteFile(@"libs\test.rb")
// Create an empty scope
engine.CreateScope()
```

### Working with Variables

When you have a scope, you can access and modify its variables.

The following code demonstrates the different methods in the `ScriptScope` class:

```
ScriptScope scope = engine.CreateScope();
// Setting a variable
scope.SetVariable("str", "IronRuby Rocks!");
// Check existence
bool b = scope.ContainsVariable("str");
// Get variable - throws an exception if variable doesn't exist
object o = scope.GetVariable("str"); // returns Object
string s = scope.GetVariable<string>("str"); // returns generic type
ObjectHandle h = scope.GetVariableHandle("str"); // returns Remoting's ObjectHandle
// Try to get variable - returns true on successful retrieval
object o1;
bool success = scope.TryGetVariable("str", out o);
string s1;
success = scope.TryGetVariable<string>("str", out s1);
ObjectHandle h1;
success = scope.TryGetVariableHandle("str", out h1);
// Get all variables
IEnumerable<KeyValuePair<string, object>> vars = scope.GetItems();
IEnumerable<string> varNames = scope.GetVariableNames();
// Remove variable
scope.RemoveVariable("str");
```

### Running with Code

As mentioned previously, all code execution methods have an optional scope parameter. This means that you can prepare a scope with parameters and pass it to the code, and the variables will be available for use.

For example, in the following code sample, I set the `str` variable on the scope and then print it from the IronRuby code:

```
scope.SetVariable("str", "IronRuby Rocks!");
engine.Execute("puts str", scope);
```

## ScriptSource

The `ScriptSource` class represents a source code. The class features, as a result, source code-related methods such as compiling, executing, or reading source code.

### Creating a ScriptSource

A `ScriptScope` can be generated via factory methods on the `ScriptEngine` instance.

`CreateScriptSourceFromFile` creates a `ScriptSource` from a code file:

```
ScriptScope scope = engine.CreateScriptSourceFromFile(@"libs\test.rb");
```

CreateScriptSourceFromString creates a ScriptSource out of a string:

```
ScriptScope scope = engine.CreateScriptSourceFromString(
    "puts 'Hello from IronRuby'");
```

### Reading the Code

One of the special capabilities the ScriptSource class has is the power to read source code files (as a whole, in batches, or line by line).

For example, remember the c:\libs\test.rb file we created earlier? Let's read it using the ScriptSource class:

```
ScriptSource source = engine.CreateScriptSourceFromFile(@"libs\test.rb");
// Get all code
source.GetCode();
// Get only line two of the code = "a + b"
source.GetCodeLine(2);
// Get lines 1-3 = add method definition
source.GetCodeLines(1, 3);
```

### Compiling the Code

If you have a piece of code that you use multiple times, it is a good idea to compile it once and then execute the compiled code every time. This might become a performance lifesaver when you have numerous calls to a single source code.

The CompileCode method compiles the code and returns a CompiledCode object that can be executed via its Execute method (even with a different scope every time):

```
ScriptSource source = engine.CreateScriptSourceFromFile(@"libs\test.rb");
CompiledCode compiled = source.CompileCode();
compiled.Execute();
```

### Executing Code

The ScriptSource class allows an interesting way of executing code. Unlike the previous ways of doing so, the ExecuteProgram method of the ScriptSource class executes the code as if it were executed from the OS command shell. It then returns the exit code representing a success or a failure:

```
ScriptSource source = engine.CreateScriptSourceFromFile(@"libs\test.rb");
int exitCode = source.ExecuteProgram();
Console.WriteLine("Script exited with code {0}", exitCode);
```

## Executing IronRuby from C#/VB.NET

The code components in the preceding section are good to know, and their capabilities might prove helpful in some advanced scenarios. However, everyday tasks are more straightforward.

In this section, I guide you through the common use cases of embedding IronRuby in a static .NET language.

### Executing an IronRuby File

If the IronRuby code is saved in an external file, you must execute it from within your C# or VB.NET code.

Listing 18.1 contains code in C# that executes the file `c:\libs\test.rb`.

---

#### LISTING 18.1 Executing an IronRuby File from C#

---

```
ScriptEngine engine = IronRuby.Ruby.CreateEngine();
engine.ExecuteFile(@"libs\test.rb");
```

---

Listing 18.2 contains code in VB.NET that executes the file `c:\libs\test.rb`.

---

#### LISTING 18.2 Executing an IronRuby File from VB.NET

---

```
Dim engine As ScriptEngine = IronRuby.Ruby.CreateEngine()
engine.ExecuteFile("C:\libs\test.rb")
```

---

### Executing IronRuby Code from a String

Executing IronRuby code from a string can be extremely helpful. The most obvious example is adding REPL (read-evaluate-print loop) capabilities to the application. All you need to do is to make it possible for the user to input IronRuby code to your application. Then all that's left for you is to execute it using the code in Listing 18.3 or Listing 18.4.

Listing 18.3 contains C# code that executes two types of IronRuby expressions, one that does not return a value and one that does.

---

#### LISTING 18.3 Executing IronRuby Code from C#

---

```
ScriptEngine engine = IronRuby.Ruby.CreateEngine();
engine.Execute("puts 'Hello'"); // Prints "Hello"
int result = engine.Execute<int>(@"def add(a, b)
                                a+b
                                end
                                add(12, 18)");
Console.WriteLine(result); // Prints 30
```

---

Listing 18.4 is equivalent to Listing 18.3, but this time it is written in VB.NET.

---

#### LISTING 18.4 Executing IronRuby Code from VB.NET

---

```
Dim engine As ScriptEngine = IronRuby.Ruby.CreateEngine()
engine.Execute("puts 'Hello'") ' Prints "Hello"
Dim result As Integer = engine.Execute(Of Integer)("def add(a, b)" & vbCrLf & _
                                                    "  a+b          " & vbCrLf & _
                                                    "end            " & vbCrLf & _
                                                    "add(12, 18)   ")

Console.WriteLine(result) ' Prints 30
```

---

## Pass Variables to and from IronRuby

When you use IronRuby in a different language, it is important to pass data between the languages. This way you can really interact between the languages and use each language's strengths to deal with the data.

The variables are set in a `ScriptScope` that can be passed to the execution method.

### USING SCOPE VARIABLES INSIDE IRONRUBY CODE

Note that inside the script you should access the scope variables with the `self` keyword. Otherwise, IronRuby considers your variables as local ones and not scope ones.

---

In Listing 18.5, I send a string to the script through a scope variable, which in turn alters it. Then I access it again from the C# code.

---

#### LISTING 18.5 Passing Variables Between C# and IronRuby

---

```
ScriptEngine engine = IronRuby.Ruby.CreateEngine();
ScriptScope scope = engine.CreateScope();
scope.SetVariable("str", "Iron");
engine.Execute("self.str = self.str + 'Ruby'", scope);
Console.WriteLine(scope.GetVariable("str")); // Prints "IronRuby"
```

---



Listing 18.6 contains the equivalent VB.NET code.

LISTING 18.6 Passing Variables Between VB.NET and IronRuby

---

```
Dim engine As ScriptEngine = IronRuby.Ruby.CreateEngine()
Dim scope As ScriptScope = engine.CreateScope()
scope.SetVariable("str", "Iron")
engine.Execute("self.str = self.str + 'Ruby'", scope)
Console.WriteLine(scope.GetVariable("str")) ' Prints "IronRuby"
```

---

## Using IronRuby Objects

Ruby has been around for a long time, and as a result there are code snippets and libraries for almost everything. Let's assume that within this load of code files you have found your one—a great looking, fun, well-documented, and feature-rich Ruby library. You remain faithful to it, and it does everything you want. But then reality strikes; you realize that you need to use it from your static language code.

The DLR, as always, saves the day and makes it possible to use IronRuby objects from C# and VB.NET. The “secret” is in the `ScriptRuntime.Globals` collection. This collection represents the global context. It is actually a `ScriptScope` instance that contains variables from the global scope. Nevertheless, the variables are not the plain old variables that you're used to; they are the objects that exist in the global scope (like modules and classes without modules).

Obtaining the module or class from the `Globals` collection is step one. The next step is to actually use it and execute methods from within it. For that, the `ScriptEngine.Operations` class comes to our aid. This class contains several different methods that run operations in the way the dynamic language does. Thus, we can create an instance of the module or class we have from the `Globals` collection.

Listing 18.7 contains an IronRuby code file with a class that we use later from .NET static languages. The file is located at `C:\libs\my_math.rb`.

LISTING 18.7 IronRuby Class for Static Language Use (my\_math.rb)

---

```
class MyMathOps
  def add(a, b)
    a + b
  end
  def multiply(a, b)
    a * b
  end
end
```

---

Listing 18.8 contains the C# code that initializes the `MyMath::Ops` class and executes its `add` and `multiply` methods.

#### LISTING 18.8 Using IronRuby Objects from C#

---

```
ScriptEngine engine = IronRuby.Ruby.CreateEngine();
ScriptScope scope = engine.ExecuteFile(@"libs\my_math.rb");

// Get the class type
object myMathOpsType = engine.Runtime.Globals.GetVariable("MyMathOps");
// Create a class instance
object myMathOps = engine.Operations.CreateInstance(myMathOpsType);

// Execute the add method
object result = engine.Operations.InvokeMember(myMathOps, "add", 1, 2);
// result = 3

// Execute the multiply method
object result2 = engine.Operations.InvokeMember(myMathOps, "multiply", "Shay", 2);
// result2 = "ShayShay"
```

---

Notice that I passed a string and a number to the `multiply` method. This is possible because of Ruby duck typing capabilities. ("`Shay`" \* 2 is a valid statement in Ruby.)

Listing 18.9 contains the same code in VB.NET.

#### LISTING 18.9 Using IronRuby Objects from VB.NET

---

```
Dim engine As ScriptEngine = IronRuby.Ruby.CreateEngine()
Dim scope As ScriptScope = engine.ExecuteFile("C:\libs\my_math.rb")
' Get the class type
Dim myMathOpsType As Object = engine.Runtime.Globals.GetVariable("MyMathOps")
' Create a class instance
Dim myMathOps As Object = engine.Operations.CreateInstance(myMathOpsType)

' Execute the add method
Dim result As Object = engine.Operations.InvokeMember(myMathOps, "add", 1, 2)
' result = 3

' Execute the multiply method
Dim result2 As Object = engine.Operations.InvokeMember(myMathOps, "multiply",
    ➤ "Shay", 2)
' result2 = "ShayShay"
```

---

## Using External Libraries

When you start to work more intensively with IronRuby, you will find yourself using more and more external libraries, like the standard library or Ruby Gems.

Listing 18.10 contains an IronRuby file, which is located at `c:\libs\base64play.rb` and takes advantage of the Base64 standard library.

LISTING 18.10 IronRuby Code That Uses the Base64 Library (`base64play.rb`)

---

```
require "base64"

encoded_string = Base64.encode64("hello")
str = Base64.decode64(encoded_string)
puts str
```

---

If we were to execute this file as we did in the “Executing an IronRuby File” section, earlier in this chapter, we would receive a “No Such File to Load — Base64” exception. This exception would occur because when you execute IronRuby code using DLR components, IronRuby doesn’t know where to look for the libraries.

All we have to do to fix that is tell the DLR where to search for libraries.

Listing 18.11 adds the IronRuby Libs folder to the runtime search paths collection and then executes the file.

LISTING 18.11 Adding Search Paths for Using External IronRuby Libraries in C#

---

```
ScriptEngine engine = IronRuby.Ruby.CreateEngine();
// Create a new collection based on the current search path collection
ICollection<string> paths = new List<string>(engine.GetSearchPaths());
// Add Ruby libraries folder
paths.Add(@"IronRuby\lib\ruby\1.8");
// Add IronRuby libraries folder
paths.Add(@"IronRuby\lib\IronRuby");
// Set the new search paths to the engine
engine.SetSearchPaths(paths);
// Execute the file
engine.ExecuteFile(@"libs\base64play.rb");
```

---

Listing 18.12 contains the equivalent code in VB.NET

---

**LISTING 18.12** Adding Search Paths for Using External IronRuby Libraries in VB.NET

---

```
Dim engine As ScriptEngine = IronRuby.Ruby.CreateEngine()  
' Create a new collection based on the current search path collection  
Dim paths As ICollection(Of String) = New List(Of String)(engine.GetSearchPaths())  
' Add Ruby libraries folder  
paths.Add("d:\IronRuby\lib\ruby\1.8")  
' Add IronRuby libraries folder  
paths.Add("d:\IronRuby\lib\IronRuby")  
' Set the new search paths to the engine  
engine.SetSearchPaths(paths)  
' Execute the file  
engine.ExecuteFile("C:\libs\base64play.rb")
```

---

## Summary

In this chapter, you learned how to run IronRuby code from C# and VB.NET. You learned about the main classes that take part in the process and read about different C#/VB.NET and IronRuby real-world scenarios.

By using IronRuby, you can, with minimal effort, extend your application. It also opens up a whole new world for CLR static languages; all Ruby libraries are now available to them (and there are many of them).

Stay tuned, this capability might become one of the killer features of the DLR in the very near future.

*This page intentionally left blank*

# PART V

## Advanced IronRuby

### IN THIS PART

CHAPTER 19 Extending IronRuby 477

*This page intentionally left blank*

## CHAPTER 19

# Extending IronRuby

### IN THIS CHAPTER

- ▶ Creating an Extension
- ▶ Using an Extension
- ▶ Building an Extension from Scratch

**I**ron Ruby provides amazing interoperability between Ruby and CLR objects. As a result, you can seamlessly use every .NET class inside IronRuby code. However, there are times when you want to write code in a static language that will act as a native Ruby object from within IronRuby code.

This is not a new idea. Ruby itself provides a way to write extensions in C: *native extensions*.

There are several reasons for that. The first reason might be to handle Win32 API calls. IronRuby can't call the Win32 API like C# or VB.Net do with `DllImport`. To solve this issue, you can either write a regular .NET assembly that does that and use it from IronRuby or write a .NET extension and use it in IronRuby as if it were native Ruby code. This becomes handy when you run into Ruby libraries that use native C extensions. C extensions are not supported by IronRuby, so to make these libraries run in IronRuby, you need to port the C extensions to .NET ones. This way you don't need to change the library Ruby code (except one `load_assembly` line), and you just replace the extension (which, from the Ruby code perspective, is the same).

Another possible reason to use .NET extensions is improving performance. IronRuby extensions use an initializer model that prevents the need of reflection when loading .NET assemblies. This way you can improve IronRuby interoperability performance and enjoy more control over the process at the same time.

The last usage reason I'll mention is the possibility to use .NET objects with Ruby-like method names. For example, when using CLR assemblies as is, you never have methods



that end with a question mark or exclamation mark (like Ruby has). With a .NET extension, you can fully control the Ruby interface.

In this chapter, you learn all about IronRuby's .NET extensions. You also learn how to create Ruby modules, classes, and methods in C# and VB.Net, and how to use them from IronRuby.Preparing Your Environment

To develop IronRuby extensions, you need an application to write the C#/VB.Net code in. It is recommended to use Visual Studio, which is the best IDE for the job.

For that matter, you can use the free edition of Visual Studio 2008, which is called Visual Studio Express.

To download the C# IDE, visit <http://www.microsoft.com/express/vcsharp>.

To download the VB.Net IDE, visit <http://www.microsoft.com/express/vb>.

You also need to have the IronRuby source code and compile it to use a tool that is not available in the product installation package. To learn how to download and compile the source of IronRuby, see Chapter 4, "Getting Started with IronRuby."

## Creating an Extension

Creating a .NET extension for IronRuby involves declaring the Ruby programming interface in a class named a *library initializer*. A library initializer contains several method calls that define the way a .NET class appears inside IronRuby. Writing the initializer code is complicated, so the IronRuby team created a tool that creates it for you. All you need is to write your code and tell the initializer generator how to work by using attributes.

This doesn't mean, however, that you can take any current .NET code, generate an initializer for it, and make it an IronRuby extension. (It is not that far from that, though.) There are some rules you need to follow to make it work.

In addition, to write extensions that work the best in IronRuby and can accept Ruby types seamlessly, you also need to be familiar with IronRuby's CLR types. For example, IronRuby doesn't use the `System.String` class, but it has a dedicated object named `MutableString`.

### EXTENSIONS IN ANY .NET LANGUAGE

Although this chapter talks about creating IronRuby extensions in C# and VB.Net, extensions are not limited to these languages only. Any .NET language that can create objects like we write in this chapter and save them into an assembly is valid for writing IronRuby extensions.

---

## Main Concepts

The IronRuby extension infrastructure is not very straightforward. You need to be familiar with its concepts and how it works to start developing IronRuby extensions. This subsection covers the basic principles for creating IronRuby .NET extensions to help you along the way.

### The Initializer Model

IronRuby extensions use a special model to avoid reflection operations that take time and reduce performance. This model is called the *initializer model*.

The initializer model has a simple concept: The extension Ruby interface is declared via special model methods. No reflection is needed in this case, and CLR objects that do not exist within the initializer aren't processed and aren't accessible via IronRuby code.

Four principles apply when creating an initializer:

- ▶ The name of the initializer class must have the next format, <namespace>LibraryInitializer. For example, if your extensions are placed in a namespace named MyIronRubyExtensions, the library initializer class name will be MyIronRubyExtensionsLibraryInitializer.
- ▶ The class must inherit from IronRuby.Builtins.LibraryInitializer.
- ▶ To declare the Ruby programming interface, you need to override the method LoadModules and write the declarations there.
- ▶ Only a single library initializer can appear per namespace. In case you have several IronRuby extensions within a single namespace, make sure to initialize them all on this namespace library initializer.

### INITIALIZER .NET CODE STRUCTURE

Because inside an initializer you actually declare every IronRuby object (modules, classes, methods, and constants), there is no restriction to the CLR code to have any special structure.

Therefore, you can write your extensions' C#/VB.Net code in multiple classes, files, or even projects. In the initializer code, you just point IronRuby to the class, method, or constant and give them their IronRuby name.

Listing 19.1 contains a sample library initializer with no code, only the structure.

#### LISTING 19.1 The Structure of an IronRuby Extension Library Initializer

```
namespace IronRubyExtension
{
    public class SampleExtensionLibraryInitializer :
        IronRuby.Builtins.LibraryInitializer
    {
        protected override void LoadModules()
        {

        }
    }
}
```

The code inside the library initializer class is complicated and requires you to be familiar with the internals of the IronRuby code. Therefore, there is a tool in the IronRuby Visual Studio solution, the class initializer generator, that generates the library code automatically according to the Ruby programming interface attributes that should decorate every IronRuby extension object.

### The Class Initializer Generator Tool

This tool reads the types and attributes in a given assembly and creates the library initializer code accordingly. The tool comes as a part of the IronRuby Visual Studio solution (Chapter 4 contains information where to obtain the IronRuby sources) and is not included in the installation package.

After you compile the solution, you can find the tool as `ClassInitGenerator.exe` in the `bin` folder of the build configuration. (For example, if you compile in Debug configuration, you can find the file under `Debug\bin`.)

The generator tool is a command-line utility. It receives the assembly path and two required arguments, as shown in Table 19.1.

TABLE 19.1 ClassInitGenerator Command-Line Arguments

| Argument   | Description  |
|--|--|
| <code>-out:[file path]</code>                      | The output file path.<br>The output file will be a C# file with the library initializer code.  |
| <code>-libraries:[semicolon separated list]</code> | A semicolon-separated list of the namespaces to investigate and generate initializer to.<br>The namespaces should exist within the given assembly. |

The full format of a class initializer call is as follows:

```
ClassInitGenerator [assembly path] -out:[output file path] -libraries:[namespace list]
```

For example, if our IronRuby extension exists within a file named `CustomExtension.dll` and the extension is written inside the namespace `CustomNamespace`, the command to create the initializer is as follows:

```
ClassInitGenerator CustomExtension.dll -out:Initializer.Generated.cs
➤ -libraries:CustomNamespace
```

After this command is executed, the library initializer class is saved to the file `Initializer.Generated.cs`.

When the class is ready, include it in your project, rebuilt it, and there you go. You have an IronRuby .NET extension ready.

### Ruby Programming Interface Attributes

Creating an IronRuby extension also involves a few CLR attributes that instruct the class initializer generator tool how to write the library initializer. These attributes do not have a role during runtime, and they are only used when the class initializer generator tool runs.

The attributes decorate .NET objects and provide the Ruby programming interface information of them. For example, there are attributes for Ruby modules, classes, methods, constants, and more.

The next sample code makes it clear that the `IronRubySample` C# class will be defined in the initializer and will appear in IronRuby code as `Sample` class:

```
[RubyClass("Sample")]
public class IronRubySample { }
```

### IronRuby .NET Extension Development Best Practices

When you write IronRuby .NET extensions, you would want to follow some recommendations. Following these practices can make the development process of IronRuby .NET extensions more clear, maintainable, and IronRuby friendly.

IronRuby .NET extension development best practices include the following:

- ▶ When throwing exceptions, use the `IronRuby.Runtime.RubyExceptions` helper class to create Ruby exceptions when possible.
- ▶ Use IronRuby CLR types when possible. For example, use `MutableString` instead of `String`.

For more information about parameter types, see the “Methods” section later in this chapter.

- ▶ IronRuby runs on .NET Framework 2.0, .NET Framework 4.0, and Silverlight. Make sure to test your extension on every IronRuby environment and apply specific changes or entirely prevent it from running on a certain environment when needed.
- ▶ Name the library initializer file `Initializer.Generated.cs`.
- ▶ Do not write IronRuby .NET extensions if you don’t really need to. Instead, you can rely on IronRuby’s interoperability capabilities and use CLR libraries directly from IronRuby code.

## The Extension Project

An IronRuby extension project is a simple class library. Therefore, to create an extension project, just create a new class library project in the language of your choice.

Three assemblies, which are located in `<IronRuby installation folder>\Bin`, must be referenced to the project if you want to use IronRuby classes, types, and attributes.

The required assemblies are IronRuby.dll, Microsoft.Scripting.dll, and Microsoft.Scripting.Core.dll.

## Target Environments

When writing an IronRuby .NET extension, keep in mind that IronRuby works in different environments: .NET 2.0, .NET 4.0, and Silverlight. Make sure that your extension runs in all different environments, or just prevent it from running in specific environments.

### IMPORTANCE OF PREPARING FOR MULTIPLE ENVIRONMENTS

Not preparing for different environments can result in unexpected behavior of the extension in the untested environments. For example, if your extension uses the file system, it can fail to run in Silverlight environment.

The best way to do it is to provide different assemblies for different environments, just like IronRuby does it. You can do so via different build configurations with compilation symbols. For more about this, see the “Building an IronRuby Extension” section later in this chapter.

## Modules

Modules in Ruby are slightly different from their equivalent in the .NET Framework, namespaces. CLR namespaces are not really objects and can’t contain methods or variables; they’re there mainly for adding order to the code. In Ruby, modules act like CLR namespaces, but they also act as mixins (see Chapter 6, “Ruby’s Code-Containing Structures,” for more about mixins), which can contain methods, variables, and more. As a result, IronRuby modules are translated to CLR classes and not to CLR namespaces.

The creation of an IronRuby module in an IronRuby extension consists of two steps: creating a CLR class and decorating it with IronRuby specific attributes.

The module attribute is `RubyModuleAttribute`, and by its very basic form, you do not need to add any arguments to it. This basic form looks as follows in code:

```
[RubyModule]
public class SampleIronRubyModule { }
```

### ATTRIBUTES ROLE

Just a reminder, attributes do not really have a runtime part in the process of creating IronRuby extensions. Their only role is to make the code understandable to the class initializer generator utility.

The `RubyModule` attribute contains several properties that describe its behavior. Table 19.2 describes these properties.

TABLE 19.2 `RubyModuleAttribute` Properties

| Name          | Description   |
|---------------|---|
| Name          | <p>The name of the module inside IronRuby. Make sure it satisfies Ruby's rules (for example, the first letter must be uppercase).</p> <p>If this argument exists in the module attribute declaration, it must come as the first argument and cannot be referenced by its name.</p>  |
| BuildConfig   | <p>Contains the compiler condition for this module. For example, if this module will not run under Silverlight, it will contain <code>"!SILVERLIGHT"</code>. If it will not run on Silverlight or .NET 2.0, the value will be <code>"!SILVERLIGHT &amp;&amp; !CLR2"</code>.</p> <p>Note that these uppercase variables are compilation symbols that you define, and they might have different names. For more about creating compilation symbols, see the "Building an IronRuby Extension" section later in this chapter.</p> <p>If this argument is omitted, it means that the module is platform independent.</p> <p>For example, the next sample indicates that the module does not work under Silverlight:</p> <pre>[RubyModule(BuildConfig = "!SILVERLIGHT")] class SampleIronRubyModule { }</pre> |
| Compatibility | <p>Indicates the Ruby version that this module is compatible with.</p> <p>The available values are <code>RubyCompatibility.Default</code>, <code>RubyCompatibility.Ruby18</code>, <code>RubyCompatibility.Ruby19</code>, <code>RubyCompatibility.Ruby20</code>.</p> <p>For example, the next sample indicates that the module works only when IronRuby runs in Ruby 1.9 mode:</p> <pre>[RubyModule(Compatibility = RubyCompatibility.Ruby19)] class SampleIronRubyModule { }</pre>  |
| DefineIn      | <p>If this module will be defined within another module (in case of a module that works as a namespace, for example), this attribute will contains the CLR type of the parent module.</p> <p>For example, the next sample has one module that is used as a namespace and another module is a part of it:</p> <pre>[RubyModule] class MyIronRubyNamespace { } [RubyModule(DefineIn = typeof(MyIronRubyNamespace))] class SampleIronRubyModule { }</pre>  |

TABLE 19.2 RubyModuleAttribute Properties

| Name           | Description  |
|----------------|--|
| Extends        | <p>If this module does not stand on its own and actually adds functionality to another module, it is considered a module extender (a bit similar to partial classes in C#). This attribute will get the type of the CLR class that represents the module that is extended by the current module class.</p> <p>For example, the next sample contains one module class and the module extender class:</p> <pre>[RubyModule] class MyIronRubyNamespace { } [RubyModule(Extends = typeof(MyIronRubyNamespace))] class SampleIronRubyModule { }</pre> |
| HideClrMembers | A Boolean value indicating whether the CLR members will be available via IronRuby.   |
| Restrictions   | <p>The restrictions the module will have. Can contain any of the <code>ModuleRestrictions</code> enum values or a combination of them (via the XOR operator, <code> </code>).</p> <p>For example, the next module will prevent its methods from being overridden and prevent name mangling:</p> <pre>[RubyModule(Restrictions = ModuleRestrictions.NoOverrides   ModuleRestrictions.NoNameMangling)] class SampleIronRubyModule { }</pre>  |

### The Basic Module Definition

The very basic module definition of an IronRuby module means that the name of the module in IronRuby is the same as the CLR class and all the other properties are using default values.

To achieve that, we decorate our class with the `RubyModule` attribute. Because this is the basic definition, there is no need to add any parameters to the attribute:

```
[RubyModule]
public class SampleIronRubyModule { }
```

### Name

The `RubyModuleAttribute` allows us to define a special IronRuby name for a module. This proves handy when you want to give a different name to your CLR object inside IronRuby.

## RUBY NAMING CONVENTIONS

Make sure to follow Ruby naming conventions when naming your Ruby modules. Ruby modules, classes, and constants must start with an uppercase letter, and method names should start with a lowercase letter.

We use the first parameter of the attribute to pass the name. The next code sample indicates that `SampleIronRubyModule` will be accessed as `SampleModule` via IronRuby code:

```
[RubyModule("SampleModule")]
public class SampleIronRubyModule { }
```

### Build Configuration

The next module attribute parameter is `BuildConfig`. This one is responsible for indicating any platform-related restrictions (for example, the restriction that the module cannot be run under Silverlight).

The attribute contains the compiler condition as the value of `BuildConfig`. The following code indicates that the module does not work under Silverlight:

```
[RubyModule(BuildConfig = "!SILVERLIGHT")]
class SampleIronRubyModule { }
```

The `SILVERLIGHT` compilation symbol is declared in the project build settings. The process of defining it is described in the “Building an IronRuby Extension” section later in this chapter.

### Compatibility

The `Compatibility` parameter indicates on which IronRuby version the module can run. The version you specify indicates the minimal version, and the extension will be considered suitable to run in higher versions, too. If this parameter is omitted, the extension is considered to be compatible with all versions.

Table 19.3 describes the available values.

TABLE 19.3 Compatibility Available Values

| Value                                  | Description   |
|--|---|
| <code>RubyCompatibility.Default</code> | The default Ruby version of IronRuby. In IronRuby version 1.0, the default is Ruby 1.8.             |
| <code>RubyCompatibility.Ruby18</code>  | Compatible with Ruby 1.8 and later. Same as not setting the <code>Compatibility</code> value.       |
| <code>RubyCompatibility.Ruby19</code>  | Compatible with Ruby 1.9 and later.   |
| <code>RubyCompatibility.Ruby20</code>  | Compatible with Ruby 2.0 and later. This is for future use since Ruby 2.0 hasn't been released yet. |

You should look at the changes in Ruby behavior between the versions to see whether your code is compatible with them. You can instruct IronRuby to run in a specific Ruby version compatibility mode by using the `-18`, `-19`, or `-20` switches with `ir.exe`. For more information about these switches, see Chapter 4.



**DefineIn**

When you want a module to be a part of another module, you need to use the `DefineIn` attribute parameter to define the CRL type of the parent module:

```
[RubyModule]
class MyIronRubyNamespace { }

[RubyModule(DefineIn = typeof(MyIronRubyNamespace))]
class SampleIronRubyModule { }
```

**Extends**

The `Extends` argument is used in two cases. The first is on CLR classes that do not exist as themselves inside IronRuby, but instead they extend another object or are being used as the Ruby programming interface of a non-IronRuby class.

The next declaration indicates that `MyNetLibOps` extends the class `MyNetLib`:

```
public class MyNetLib { }

[RubyModule("MyNetLib", Extends = typeof(MyNetLib))]
public class MyNetLibOps { }
```

The second case is when an IronRuby module already exists and your new module only adds to it. For example, the next declaration indicates that `MyNetLibOps` extends IronRuby's `Kernel` module:

```
[RubyModule(Extends = typeof(IronRuby.Builtins.Kernel))]
public class MyNetLibOps { }
```

This statement generally indicates that the CLR class itself will not have any meaning and all IronRuby is going to care about is the class internal members that will be added to a different defined module.

**Restrictions**

The restrictions are used to apply special behavior to the module members after they are interpreted by IronRuby. The `Restrictions` attribute value is a combination of `ModuleRestrictions` enum values.

Table 19.4 describes possible restriction values.

The restrictions declaration via the `RubyModule` attribute is done via the `Restrictions` parameter. In the next sample, `SampleIronRubyModule` is declared with the `NoOverrides` and `NoNameMangling` restrictions:

```
[RubyModule(Restrictions = ModuleRestrictions.NoOverrides |
            ModuleRestrictions.NoNameMangling)]
class SampleIronRubyModule { }
```

TABLE 19.4 ModuleRestrictions Enum Values

| Value  | Description  |
|--|--|
| <code>ModuleRestrictions.NotPublished</code>   | The module will not be published on IronRuby's global context.<br>To access it, you must provide a way manually (like setting a constant in a class that references the module). |
| <code>ModuleRestrictions.NoOverrides</code>    | Overriding the module methods will not be available.   |
| <code>ModuleRestrictions.NoNameMangling</code> | Method names will not be mangled.  |
| <code>ModuleRestrictions.Builtin</code>        | A combination of all the above. Used for built-in modules.   |
| <code>ModuleRestrictions.All</code>            | Same as <code>ModuleRestrictions.Builtin</code> .  |
| <code>ModuleRestrictions.None</code>           | None of the above; no special restrictions. This is the default value.   |

### Hide CLR Members

When your CLR IronRuby module contains methods, these methods will be interpreted when the module is loaded and will become available to the IronRuby code. If the .NET code you write should not be exposed to IronRuby code (except the parts that are explicitly decorated with IronRuby attributes), you need to signal IronRuby to hide the CLR members.

To declare this behavior, the `RubyModule` attribute provides the `HideClrMembers` Boolean value:

```
[RubyModule(HideClrMembers = true)]
class SampleIronRubyModule { }
```

### Mixin Definitions

Your module can include code of other IronRuby types or mix in other modules to add features to it. For example, by mixing the `IronRuby.Builtins.Comparable` class, you will add comparison capabilities between the objects of your module.

This is done using `IncludesAttribute`. Along with the `RubyModuleAttribute` definition, `IncludesAttribute` adds the definition of the mixin classes. The attribute can be added to modules and classes.

The attribute receives two parameters: a list of types that represent the types of the mixin classes and an optional Boolean parameter named `Copy` that indicates whether the types should be copied into the class or be used as mixin. The default of `Copy` is `false`.

The next code adds the module class `SampleIronRubyMixin` as a mixin to the `SampleIronRubyModule` class:

```
[RubyModule]
class SampleIronRubyMixin { }

[RubyModule]
[Includes(typeof(SampleIronRubyMixin), Copy = false)]
class SampleIronRubyModule { }
```

To include more than one class, just pass the types one after another. The next sample includes the built-in `Comparable` and `Enumerable` mixins in the `SampleIronRubyModule` class:

```
[RubyModule]
[Includes(typeof(IronRuby.Builtins.Comparable),
          typeof(IronRuby.Builtins.Enumerable))]
class SampleIronRubyModule { }
```

## USING BUILT-IN MIXINS

In certain scenarios, you need to use IronRuby built-in mixin modules like `Comparable` or `Enumerable`.

To do that, you need to find out the type name. To do so, I suggest you to go through the `IronRuby.Builtins` namespace. It contains all the built-in types. A quick look there and you can locate the type you need.

Notice that you have to add a reference to the `IronRuby.Libraries.dll` assembly to use the `Comparable` class and other built-in classes.

## Classes

IronRuby classes are similar to .NET classes. They contain methods, constants, and variables and can even contain modules and classes inside them (although this is uncommon).

Just like IronRuby modules, creating an IronRuby class consists of defining the class and decorating it with a related attribute.

From Ruby's perspective, classes inherit from modules. They get all the module characteristics and add some of their own. This behavior is transferred to IronRuby, too, and as a result, `RubyClass` inherits from `RubyModule`. Therefore, most of the IronRuby class definition process is identical to the module one.

The attribute that should decorate .NET classes that are intended to be IronRuby classes is `RubyClassAttribute`. This attribute contains the same parameters as `RubyModuleAttribute`, with an extra one. Read the previous section to learn about all the `RubyModuleAttribute` parameters, which can also be used with `RubyClassAttribute`.

Table 19.5 describes the `RubyClassAttribute` unique properties.

TABLE 19.5 RubyClassAttribute Unique Properties (see also Table 19.2)

| Property | Description  |
|----------|--|
| Inherits | If this class inherits from another IronRuby class, this property contains the .NET type of the IronRuby superclass. |

### The Basics

The basic definition of a class is done by using the `RubyClassAttribute` without any parameters:

```
[RubyClass]
public class MyIronRubyClass { }
```

By doing so, you define a class that will be called by its CLR name from within IronRuby code, and all other properties use their default values (or no value at all).

Note that when your IronRuby class exists within another class, which is an IronRuby class or module, too, the name of your class will be `<parent class name>::<class name>`. For example, the next code generates an IronRuby module and a class:

```
[RubyModule("SampleModule")]
public class SampleIronRubyModule
{
    [RubyClass("SampleClass")]
    public class SampleIronRubyClass { }
}
```

Inside IronRuby code, to access `SampleClass` you need to refer it as `SampleModule::SampleClass`.

### Inherits

In case the IronRuby class you're writing should inherit from another IronRuby class (built in or custom), you should indicate its type on the `Inherits` parameter. If your CLR class really inherits from the parent CLR class, you can omit this parameter, and the initializer generator will make the connection automatically.

The next code sample makes `SampleIronRubyClass` inherit from `SampleIronRubyParentClass`:

```
[RubyClass]
public class SampleIronRubyParentClass { }
[RubyClass(Inherits = typeof(SampleIronRubyParentClass))]
public class SampleIronRubyClass { }
```

This preceding code sample is equivalent to the following:

```
[RubyClass]
public class SampleIronRubyParentClass { }
[RubyClass]
public class SampleIronRubyClass : SampleIronRubyParentClass { }
```

For a description of all other properties of `RubyClassAttribute`, look at the properties of its parent class, `RubyModuleAttribute`, previously in this chapter.

### Singleton Classes

Ruby supports defining singleton classes. You can achieve the same behavior by attributes when you build your own IronRuby .NET class.

To do so, instead of using `RubyClassAttribute`, use `RubySingletonAttribute`. This attribute inherits from `RubyModuleAttribute` and therefore does not contain the `Inherits` parameter of the `RubyClass` attribute.

However, this is not enough. As a result of the class being singleton, you must define where the singleton instance is stored. You do so inside an `IronRuby` constant, and the way to define it is to use `RubyConstantAttribute` on the class, as well.

The next code sample declares `SampleIronRubyClass` as a singleton class:

```
[RubyConstant]
[RubySingleton]
public class SampleIronRubyClass { }
```

`RubySingletonAttribute` receives the same parameters as `RubyModuleAttribute`. (You can review the `RubyModuleAttribute` parameters by looking back over what we've already covered in this chapter.)

### Exception Classes

`IronRuby` provides several exception classes. All of them exist within the `IronRuby.Builtins` namespace, and you will spot them by the “Error” suffix (like `NoMemoryError` or `EOFError`). The `RubyExceptions` helper class also makes creating `IronRuby` exceptions easier.

However, sometimes the available classes aren't answering your needs, and you want to define a new `IronRuby` exception class.

`IronRuby` exception classes should follow these rules:

- ▶ The class should inherit from the `Exception` class or one of its derived classes.
- ▶ The class should be serializable. This can be achieved by using `SerializableAttribute` and making sure all public members of the class are also serializable.
- ▶ The name of the class should end with “Error” to meet Ruby's conventions.

- The class must have a constructor that receives a message and an inner exception objects.
- The exception class should be decorated using `RubyExceptionAttribute`.

`RubyExceptionAttribute` receives the same parameters as `RubyClassAttribute`. There is one difference, though: You must specify its IronRuby name.

The next code sample defines a custom IronRuby exception class named `CustomIronRubyError`:

```
[RubyException("CustomIronRubyError")]
[Serializable]
public class CustomIronRubyError : Exception
{
    public CustomIronRubyError(string message, Exception inner) : base(message, inner)
    { }
}
```

### Undefining Class Methods

IronRuby attributes allow you to undefine methods that are defined within the class. This technique is used primary to prevent classes from being created via the new method. You can then supply a different method to do that or prevent creating class instances from IronRuby at all (and leave it possible only to the .NET extension code).

This is done via `UndefineMethodAttribute`, which receives two parameters. The first is the name of the method. The second one is an optional Boolean parameter named `IsStatic`, which indicates whether the method is an instance or a class method.

For example, the next code sample prevents IronRuby code from creating instances of the `SampleIronRubyClass` class by undefining the new method (which is a class method):

```
[RubyClass]
[UndefineMethod("new", IsStatic = true)]
public class SampleIronRubyClass { }
```

## Methods

Every class or module in Ruby can contain methods. Methods are actually the place where the logics of the application can be written.

Methods that are supposed to be used as IronRuby methods must be declared as static and should also be decorated with `RubyMethodAttribute`. Table 19.6 describes the properties of `RubyMethodAttribute`.

TABLE 19.6 RubyMethodAttribute Properties

| Property         | Description   |
|------------------|---|
| Name             | The name of the method inside IronRuby code.<br>Required. Must be passed as the first parameter.  |
| methodAttributes | The method characteristics, such as its visibility (private, protected, public) or its context (instance method or class method).<br>Optional. If declared, must be passed as the second parameter. |
| BuildConfig      | Contains the compiler condition for this method. See Table 19.2 for more information about this property.   |
| Compatibility    | Indicates the Ruby version that this method is compatible with.<br>See Table 19.2 for more information about this property.   |

### Special Parameters

Every IronRuby method receives a few parameters besides the real method parameters. These parameters are sent to the method automatically and are not available in IronRuby code.

Table 19.7 contains the parameters in the order of their definition and further details about each of them. Note that the order of the parameters is essential and may not be changed.

TABLE 19.7 IronRuby Method Special Parameters (in order)

| Type                     | Possible Appearances | Description   |
|--------------------------|----------------------|---|
| CallSiteStorage          | 0 or more            | If the library needs to call back into Ruby code, the needed call site storages can be passed here.<br>For call site storage types, look in the IronRuby code at <code>Ruby\Runtime\CallSiteStorages.cs</code>  |
| RubyContext or RubyClass | 0 or 1               | A <code>RubyContext</code> object will be available for class methods and will provide a way to access information about the context (for example, global variables or several environment options).<br>A <code>RubyClass</code> object will be available for instance methods and will provide a way to access the class (getting all its mixins, adding methods, and more). |
| BlockParam               | 0 or 1               | If the method receives a Ruby block, this parameter will contain it and enable to get information about it and invoke it.   |

TABLE 19.7 IronRuby Method Special Parameters (in order)

| Type                | Possible Appearances | Description  |
|---------------------|----------------------|--|
| Object or RubyClass | 1                    | Required and must be named <code>self</code> .<br>For instance methods, the type of this parameter will be the class itself, providing a way to access the class instance members.<br>For class methods, the type of this parameter will be <code>RubyClass</code> , providing a way to access the class object and the class members. |
| Others              | 0 or more            | All other parameters and their types.  |

For example, the next sample is an IronRuby class method that from IronRuby code doesn't require any parameters except from a block. However, it does receive several ones that are filled automatically by the IronRuby interpreter:

```
[RubyMethod("sample", RubyMethodAttributes.PublicSingleton)]
public static void Sample(RubyContext context, BlockParam block, RubyClass self)
{
}
```

### The Basics

The basic definition for IronRuby methods includes their IronRuby name. The name, unlike for IronRuby classes and modules, is required because of the difference in naming conventions between Ruby methods and CLR methods. The naming convention for Ruby methods is all lowercase, with underscore as a word delimiter.

The next code sample contains a definition of an IronRuby extension class with a single method that does not receive parameters in IronRuby and prints a "Hello World!" message:

```
[RubyClass]
public class SampleIronRubyClass
{
    [RubyMethod("say_hello")]
    public static void SayHello(SampleIronRubyClass self)
    {
        Console.WriteLine("Hello World!");
    }
}
```



From IronRuby, after the extension is loaded, the method will be used as follows:

```
sample = SampleIronRubyClass.new
sample.say_hello # Prints "Hello World!"
```

### Method Attributes

The second parameter of `RubyMethodAttribute` is the method attributes. This parameter helps defining the way this method is declared. The value of the parameter is one of the `RubyMethodAttributes` enum values or a combination of several values (via the XOR operator). Table 19.8 describes the available values.

TABLE 19.8 `RubyMethodAttributes` Values

| Value                           | Description   |
|---------------------------------|---|
| <code>Public</code>             | The method is a public method.  |
| <code>Protected</code>          | The method is a protected method.   |
| <code>Private</code>            | The method is a private method.   |
| <code>DefaultVisibility</code>  | Same as <code>Public</code> .   |
| <code>Empty</code>              | Indicates that the method does nothing.   |
| <code>Instance</code>           | The method is an instance method.   |
| <code>Singleton</code>          | The method is a class method.   |
| <code>NoEvent</code>            | Do not trigger <code>method_added</code> metaprogramming method when the method is defined.                             |
| <code>PublicInstance</code>     | The method is a public instance method.<br>A combination of <code>Public</code> and <code>Instance</code> values.       |
| <code>PrivateInstance</code>    | The method is a private instance method.<br>A combination of <code>Private</code> and <code>Instance</code> values.     |
| <code>ProtectedInstance</code>  | The method is a protected instance method.<br>A combination of <code>Protected</code> and <code>Instance</code> values. |
| <code>PublicSingleton</code>    | The method is a public class method.<br>A combination of <code>Public</code> and <code>Singleton</code> values.         |
| <code>PrivateSingleton</code>   | The method is a private class method.<br>A combination of <code>Private</code> and <code>Singleton</code> values.       |
| <code>ProtectedSingleton</code> | The method is a protected class method.<br>A combination of <code>Protected</code> and <code>Singleton</code> values.   |

TABLE 19.8 RubyMethodAttributes Values

| Value          | Description  |
|----------------|--|
| ModuleFunction | The method is a public method declared as an instance and a class method.<br>A combination of Public, Instance and Singleton values. |
| Default        | Same as PublicInstance.  |

The next sample defines the SayHello method as a public class method:

```
[RubyClass]
public class SampleIronRubyClass
{
    [RubyMethod("say_hello", RubyMethodAttributes.PublicSingleton)]
    public static void SayHello(RubyClass self)
    {
        Console.WriteLine("Hello World!");
    }
}
```

From IronRuby code, the method will now be available as a class method:

```
SampleIronRubyClass.say_hello # Prints "Hello World!"
```

For more about BuildConfig and Compatibility, see the “Modules” section earlier in this chapter.

### Parameters

Method parameters have a few regulations you should be aware of.

First and foremost, always prefer IronRuby types to CLR ones. This will prevent unneeded conversions between the types and improve performance. For example, prefer `MutableString` to `System.String` or `RubyArray` to `System.Array`. Take a look at the types available in the `IronRuby.Builtins` namespace to discover the available types.

Second, a few attributes can decorate the method parameters and enforce different behaviors, as shown in Table 19.9.

TABLE 19.9 Parameter IronRuby Attributes

| Attribute       | Description   |
|-----------------|---|
| DefaultProtocol | Enforces the parameter to contain its expected type.<br>For example, for <code>MutableString</code> parameters, if a <code>System.String</code> is passed, IronRuby will convert it to <code>MutableString</code> automatically when <code>DefaultProtocol</code> exists. |

TABLE 19.9 Parameter IronRuby Attributes

| Attribute             | Description   |
|-----------------------|---|
| NotNull               | Indicates that the parameter is required.   |
| NotNullItems          | Can decorate array parameters. Indicates that the array items cannot be null values.      |
| Optional              | Indicates that the parameter is optional.   |
| DefaultParameterValue | Indicates the default parameter value. This means that the parameter is optional, too as. |

For example, the next method uses the several different attributes for its parameters:

```
[RubyMethod("sample_method")]
public static void SampleMethod(object self, [DefaultProtocol]MutableString str,
                                           [DefaultProtocol]int num,
                                           [NotNull]MutableString nonNullStr,
                                           [NotNull, NotNullItems]int[] numbers,
                                           [Optional]object param,
                                           [DefaultParameterValue(true)]bool yesOrNo)
{
}
```

The last detail you need to know about the Ruby method parameters is Ruby's special parameters—blocks, procs, lambdas, and array arguments:

- **Blocks:** When a method needs to receive a block, this is done via the `BlockParam` parameter. See the “Special Parameters” section to see where this parameter should be placed. If your method requires a block, make sure to use the `NotNullAttribute` on it.

For example, the next method receives a block and executes it:

```
[RubyMethod("sample", RubyMethodAttributes.PublicSingleton)]
public static void Sample([NotNull]BlockParam block, RubyClass self)
{
    // The result variable will contain the return value of the block
    object result;
    // Execute the block and pass it a "Hello" string
    block.Yield("Hello", out result);
}
```

- **Procs and lambdas:** To get a proc or lambda object as a parameter, use the `IronRuby.Builtins.Proc` type for the parameter. Then use the `Call` method to execute it. To identify whether the parameter value is a proc or lambda, you can use the `Proc.Kind` property.

The next code sample gets a proc as its parameter and throws an exception if the value is not a lambda object and not a proc:

```
[RubyMethod("sample")]
public static void Sample(object self, Proc proc)
{
    if (proc.Kind == ProcKind.Lambda)
    {
        RubyExceptions.CreateArgumentError("Lambdas are forbidden");
    }
    proc.Call();
}
```

- **Array argument:** Ruby's array argument means that the method can receive an unlimited number of parameters. This is done, via Ruby code, with the asterisk (\*). To achieve that in your .NET extension, you need to use the equivalent of the array argument in your .NET language. In C#, this will be the `params` keyword. Note that this argument must be the last one.

For example, the following code sample allows an arbitrary number of arguments to be passed to the method:

```
[RubyMethod("sample")]
public static void Sample(object self, params object[] args)
{
}
```

### Constructor Methods

When you want to define constructor behavior to your IronRuby class, you need to decorate the constructor method with `RubyConstructorAttribute`. Because IronRuby CLR methods must be static methods, the constructor will be a regular IronRuby method that creates the requested type. In other words, methods decorated with the `RubyConstructor` attribute are factory methods.

`RubyConstructorAttribute` has two optional parameters: `BuildConfig` and `Compatibility`. For more information about them, see the “Modules” section earlier in this chapter.

The following code sample defines a constructor to the `SampleIronRubyClass` class, which initializes the class and returns it:

```
[RubyClass]
public class SampleIronRubyClass
{
    [RubyConstructor]
    public static SampleIronRubyClass CreateInstance(RubyClass self)
```

```

{
    return new SampleIronRubyClass();
}
}

```

## THE INITIALIZE METHOD

You can define an `initialize` method in your code, which is Ruby's constructor method. However, this method will not be called automatically, and you will have to call it from the method you define as a Ruby constructor.

This behavior is the result of the real behavior of the Ruby language: The `initialize` method doesn't really construct an object, it only initializes it.

### Ruby Attributes

Many Ruby classes contain attributes, which are similar to CLR properties. However, Ruby doesn't really have attributes. The attribute creator methods `attr_accessor`, `attr_reader`, and `attr_writer` actually create methods that emulate real attributes. For example, if you declare a reader-writer attribute named `full_name`, two methods will be created automatically: `full_name` and `full_name=`.

If you want to declare attributes in your IronRuby extension, you must define a getter and a setter method for each attribute. Be aware of the difference between instance attributes and class attributes.

Instance attribute values can be saved within instance members. Class attribute values, on the other hand, should be saved in class members.

The following sample defines two attributes; one is an instance attribute, and one is a class attribute:

```

[RubyClass]
public class Test
{
    // Instance attribute
    public MutableString InstanceAttributeValue;

    [RubyMethod("instance_attribute")]
    public static MutableString GetInstanceAttribute(Test self)
    {
        return self.InstanceAttributeValue;
    }

    [RubyMethod("instance_attribute=")]
    public static void SetInstanceAttribute(Test self, [NotNull]MutableString value)
    {
        self.InstanceAttributeValue = value;
    }
}

```

```
// Class attribute
[RubyMethod("class_attribute", RubyMethodAttributes.PublicSingleton)]
public static MutableString GetClassAttribute(RubyClass self)
{
    object value;
    if (self.TryGetClassVariable("class_attribute_value", out value))
    {
        return (MutableString)value;
    }
    return MutableString.CreateEmpty();
}
[RubyMethod("class_attribute=", RubyMethodAttributes.PublicSingleton)]
public static void SetClassAttribute(RubyClass self, [NotNull]MutableString value)
{
    self.SetClassVariable("class_attribute_value", value);
}
}
```

### Aliasing Methods

In Ruby, you can give methods an alias name. You can also do so in your IronRuby .NET extension code.

There are two possible ways to do that. The first is to add another `RubyMethodAttribute` declaration with the alias name:

```
[RubyMethod("sample")]
[RubyMethod("another_sample")]
public void Sample(object self) { }
```

The second way is to use the `AliasMethodAttribute` on the class definition. The first argument of the attribute is the alias name, and the second attribute is the current name:

```
[RubyClass]
[AliasMethod("another_sample", "sample")]
public class SampleIronRubyClass
{
    [RubyMethod("sample")]
    public void Sample(object self) { }
}
```

There is no actual difference between these ways. Choosing one over the other consists of where you want the method alias to be—above the method declaration or above the class declaration.

**Hiding Methods**

When your IronRuby class copies another IronRuby class or module (by using `IncludesAttribute` and `Copy = true`), there might be some methods that you do not want to include in your class.

IronRuby attributes provide a solution to this need through `HideMethodAttribute`. This attribute decorates the class and has two parameters. The first is required and must appear as the first argument. It contains the name of the method to hide. The second parameter is named `IsStatic` and indicates whether the method to hide is an instance method or a class method. The default is instance method.

The following code sample hides the method `print` from the `SampleIronRubyClass` class:

```
[RubyModule]
public class SampleIronRubyModule
{
    [RubyMethod("print")]
    public static void Print(object self)
    {
        Console.WriteLine("Print");
    }
}

[RubyClass]
[Includes(typeof(SampleIronRubyModule), Copy = true)]
[HideMethod("print")]
public class SampleIronRubyClass
{
}
```

**Hiding Methods from Ruby's Stack Trace**

In rare cases, you want your method to not appear in the stack trace. IronRuby has a small number of methods that use this behavior (for example, `method_missing` and `raise`).

`RubyStackTraceHiddenAttribute` is used to hide methods from the stack trace.

For example, the next method does not appear in Ruby's stack trace:

```
[RubyMethod("sample")]
[RubyStackTraceHidden]
public void Sample(object self) { }
```

## Constants

IronRuby extension classes can also contain constants. Defining a constant is done by decorating a CLR constant with `RubyConstantAttribute`.

`RubyConstantAttribute` receives an optional IronRuby name for the constant. If this parameter is omitted, the CLR name is used.

The following code defines an IronRuby constant:

```
[RubyConstant("MyConstant")]  
public const int MyIronRubyConstant = 1;
```

## Using an Extension in IronRuby

Using an extension inside IronRuby is simple. With all the hard work behind us, all we have to do is load the CLR assembly and use it as if it were a regular Ruby object.

Loading a .NET extension is done via the `load_assembly` method. This method, commonly receiving only a single parameter, needs two parameters to load our extension.

The first one is the assembly name, which can be a partial or a strong name (see Chapter 9, “NET Interoperability Fundamentals”).

The second one is the namespace of our library. IronRuby will then go and look for the library initializer class according to the namespace name.

For example, our extension assembly is located within the current folder, and its name is `MyIronRubyExtensions.dll`. In addition, this assembly has a namespace called `IronRubyExtensions` that contains a library initializer. This is the way how we load it:

```
load_assembly "MyIronRubyExtensions", "IronRubyExtensions"
```

After this line, our IronRuby .NET extension is loaded and can be used from IronRuby code just as if it were written in native Ruby code.

## Building an IronRuby Extension

Let's use everything we've learned in this chapter and create our own IronRuby .NET extension.

The extension we are going to write is simple. It will get a text message and display it in a couple of ways. Another feature will be to display the message in color.

Even though the extension is simple, it can give basic experience (and a taste for) writing real IronRuby .NET extensions.



## Creating the Extension Visual Studio Project

First we need to create a project for the extension. Follow these steps to create and prepare a project for writing an IronRuby extension:

1. Open Visual Studio and go to File > New > Project.
2. Choose your .NET programming language from the left panel and Class Library from the right panel, as shown in Figure 19.1. Set the directory and project name and click OK.

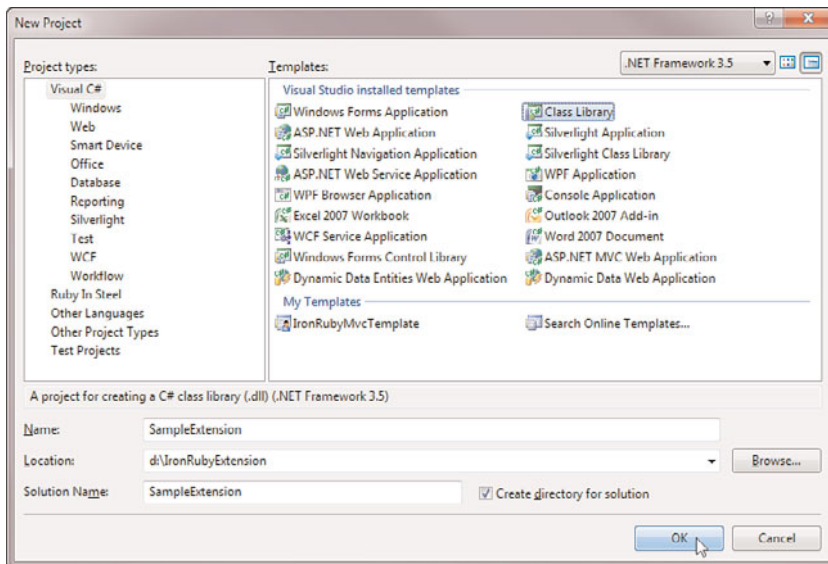


FIGURE 19.1 The New Project window when creating an IronRuby .NET extension project.

3. After the project is created, go to Project > Add Reference.
4. In the Add Reference dialog, display the Browse tab and navigate to <IronRuby installation folder>\Bin.
5. Select three files (you can do that by holding Ctrl pressed during the selection): IronRuby.dll, Microsoft.Scripting.dll, and Microsoft.Scripting.Core.dll. Click OK to apply.

You're now ready to start developing the IronRuby .NET extension.

## Adding Build Configurations

As previously mentioned in this chapter, IronRuby runs in different environments, and you need to make sure that your extension is compatible in all of them. It is okay to not support an environment, but make sure to let the user know about it right away to keep the unexpected behavior to a minimum.

You can use Visual Studio features to create different versions of your extension, one for every environment. This is controlled, inside Visual Studio, by the Configuration Manager, where you can add build configurations. Every project can be assigned compilation symbols for each build configuration. With this compilation symbol, you can make a block of code to compile or not to compile when a specific build configuration is selected.

The IronRuby team has six build configurations in the IronRuby project, which are good to incorporate to your .NET extension project, too: .NET Framework 2.0 Debug and Release, .NET Framework 4.0 Debug and Release, and Silverlight Debug and Release.

The following steps take you through the process of adding a build configuration for Silverlight and a matching compilation symbol named SILVERLIGHT:

1. When the IronRuby .NET extension solution is opened in Visual Studio, go to the Build > Configuration Manager.
2. The Configuration Manager dialog will open. Click the upper-left list box and choose <New...>.
3. In the New Solution Configuration dialog, enter **Silverlight** in the Name field as shown in Figure 19.2 and click OK.

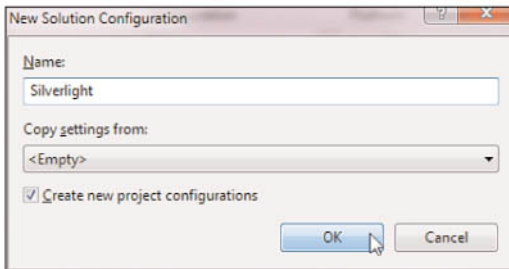


FIGURE 19.2 Adding a SILVERLIGHT compilation symbol.

4. Click Close to close the Configuration Manager.
5. Click View > Solution Explorer.
6. A panel appears with a tree of all the projects and files in the solution. Click once on the project name to select it as shown in Figure 19.3.

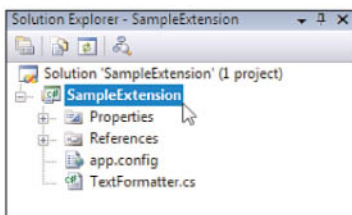


FIGURE 19.3 Solution Explorer presenting the extension project file tree.

7. In the menu, go to Project > SampleExtension Properties.
8. A view of the project properties will open. On the left navigation panel, click Build.
9. On the right, select Silverlight in the Configuration select box.
10. In the Conditional Compilation Symbols field, enter **SILVERLIGHT**. This is where you define the compilation symbols for the specific build. It is a semicolon-separated list. Figure 19.4 shows how it will look when you finish.

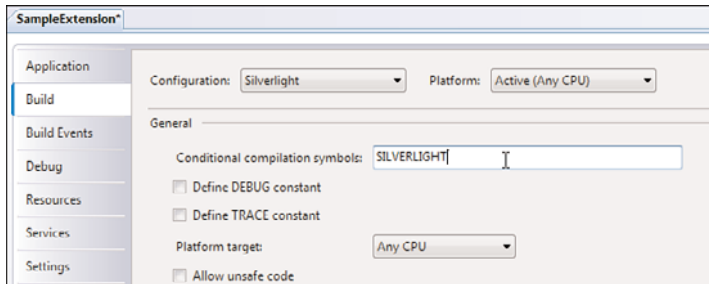


FIGURE 19.4 The project configuration with a compilation symbol defined.

11. Go to File > Save All to save the new build configuration.

With these steps, you can define more build configurations and more compilation symbols.

Note that when you define build configurations, you must compile the solution in each configuration (and therefore have a different assembly for each configuration).

IronRuby code contains three build configurations that are presented in Table 19.10. You can use these configurations as a guideline to the required supported environments for your project.

TABLE 19.10 IronRuby Build Configurations

| Compilation Symbol | Description                         |
|--------------------|-------------------------------------|
| CLR2               | Targeted to .NET 2.0                |
| CLR4               | Targeted to .NET 4.0                |
| SILVERLIGHT        | Targeted to Silverlight environment |

## Creating the Actual Code

Before I start to write the Ruby programming interface, I'd like to write the actual library code. Because it is not convenient to use static methods only, I recommend that you write your code as you are used to writing it and then add static methods that provide the Ruby programming interface. This is exactly how I'm going to build this sample extension.

Listing 19.2 contains the code of the class without any extension attributes or methods, just plain C#.

---

LISTING 19.2 Library Code Without IronRuby Extension Methods

---

```
using System;

namespace MyRubyExtension
{
    public class TextFormatter
    {
        public MutableString Text { get; set; }
        public bool UseColors { get; set; }

        public void Congratulate()
        {
            ConsoleColor originalColor = Console.ForegroundColor;
            if (UseColors)
            {
                Console.ForegroundColor = ConsoleColor.Green;
            }
            Console.WriteLine("Congratulations!");
            Console.WriteLine(Text);
            if (UseColors)
            {
                Console.ForegroundColor = originalColor;
            }
        }

        public void Warn()
        {
            ConsoleColor originalColor = Console.ForegroundColor;
            if (UseColors)
            {
                Console.ForegroundColor = ConsoleColor.Red;
            }
            Console.WriteLine("Beware!");
            Console.WriteLine(Text);
            if (UseColors)
            {
                Console.ForegroundColor = originalColor;
            }
        }
    }
}
```

---

As you can see in Listing 19.2, the code is quite simple. All it does is set the console colors if needed and write the given text with a related header.

## Creating the Ruby Programming Interface

Now we want to turn our very regular C# class into an IronRuby .NET extension class. I want to use the class as is, so I just add a part to it that redirects IronRuby requests to the real CLR instance methods.

The code has three more interesting parts. First, because I use the console, this code does not work in Silverlight. To cope with that, I add a no-Silverlight condition to the class `BuildConfig` condition. This is only for the sample purposes, because in this case, my IronRuby extension in Silverlight mode will be empty. If your extension doesn't work in Silverlight, just let the users know about it before they download it and don't provide them an empty assembly.

The second interesting part is the IronRuby constructor. This changes the way the CLR class works: Instead of using attributes for the `Text` and `UseColors` CLR properties, I chose to send them via the constructor only. This is the kind of freedom you get when developing IronRuby .NET extensions; you are not tied to the CLR implementation, and you can provide IronRuby code a more Ruby-esque look and feel.

Third, take a look how I alias the `warn` method. Although C# doesn't allow me to use exclamation mark in method names, Ruby allows that, and with the IronRuby extension attributes I can give my constrained CLR methods a Ruby name that fits it the most.

Listing 19.3 contains the class with its Ruby programming interface part.

### LISTING 19.3 Sample .NET Extension Full Code

---

```
using System;
using IronRuby.Builtins;
using IronRuby.Runtime;
using Microsoft.Scripting.Runtime;
using System.Runtime.InteropServices;

namespace MyRubyExtension
{
    [RubyClass(BuildConfig = "!SILVERLIGHT", HideClrMembers = true)]
    public class TextFormatter
    {
        public MutableString Text { get; set; }
        public bool UseColors { get; set; }

        public void Congratulate()
        {
            ConsoleColor originalColor = Console.ForegroundColor;
            if (UseColors)
```

```
{
    Console.ForegroundColor = ConsoleColor.Green;
}
Console.WriteLine("Congratulations!");
Console.WriteLine(Text);
if (UseColors)
{
    Console.ForegroundColor = originalColor;
}
}

public void Warn()
{
    ConsoleColor originalColor = Console.ForegroundColor;
    if (UseColors)
    {
        Console.ForegroundColor = ConsoleColor.Red;
    }
    Console.WriteLine("Beware!");
    Console.WriteLine(Text);
    if (UseColors)
    {
        Console.ForegroundColor = originalColor;
    }
}

// ---- Ruby Programming Interface ---
[RubyConstructor]
public static TextFormatter CreateTextFormatter(RubyClass self,
                                                [DefaultProtocol, NotNull]MutableString text,
                                                [DefaultParameterValue(true)]bool useColors)
{
    TextFormatter instance = new TextFormatter();
    return Initialize(instance, text, useColors);
}

[RubyMethod("initialize", RubyMethodAttributes.PrivateInstance)]
public static TextFormatter Initialize(TextFormatter self,
                                      [NotNull]MutableString text, bool useColors)
{
    self.Text = text;
    self.UseColors = useColors;
    return self;
}
```

```

[RubyMethod("congratulate", RubyMethodAttributes.PublicInstance)]
public static void IronRubyCongratualate(TextFormatter formatter)
{
    formatter.Congratulate();
}

[RubyMethod("warn", RubyMethodAttributes.PublicInstance)]
[RubyMethod("warn!", RubyMethodAttributes.PublicInstance)]
public static void IronRubyWarn(TextFormatter formatter)
{
    formatter.Warn();
}
}
}

```

---

## Generating the Library\_INITIALIZER

Now that we have our extension code ready, we need to add the initializer code to it.

First, compile the code. Make sure that Debug is the current selected build configuration. (Do that from the Configuration Manager window by selecting the active build configuration in the upper-left select box.)

After the project is built, we need to create the initializer code. This is done with the ClassInitGenerator.exe utility. Navigate to the tool location. Assuming that the built extension assembly exists in C:\SampleExtension\SampleExtension\bin\Debug; this is the command that should be executed:

```

classinitgenerator "C:\
SampleExtension\SampleExtension\bin\Debug\SampleExtension.dll" -out:" C:\ SampleEx-
tension\SampleExtension\Initializer.Generated.cs" -libraries:MyRubyExtension

```

The output will be a single line:

```
Library MyRubyExtension
```

For more about this tool, see the “The Class Initializer Generator Tool” section earlier in this chapter.

### WHAT TO DO WHEN THE GENERATED\_INITIALIZER IS EMPTY

When you have multiple IronRuby assemblies on your computer (like the compiled sources and the official release), you might run into a problem when the generated initializer file contains no actual code (only the class structure).

A problem like that can happen when the class initializer generator is built with a different IronRuby.dll assembly than the extension project. They both must reference the same IronRuby.dll assembly because otherwise the initializer generator can't locate the custom attributes it's looking for.

---

With the file in the project directory, we just need to include it in our project and recompile. Follow the next steps to do that:

1. Go to View > Solution Explorer.
2. In the Solution Explorer panel, click the project node of the tree to select it, and then click the Show All Files icon.

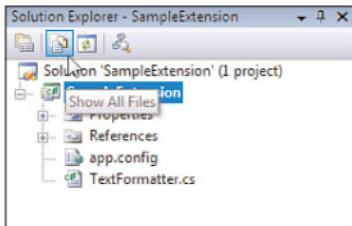


FIGURE 19.5 Show All Files icon in Solution Explorer.

3. A few directories and the initializer files will be added to the tree. Right-click the `Initializer.Generated.cs` file and choose Include in Project.

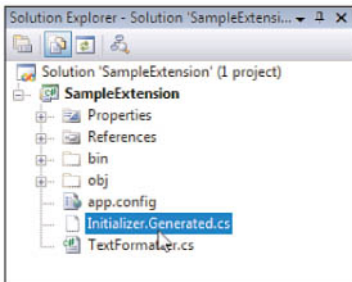


FIGURE 19.6 Choosing the initializer file in Solution Explorer.

4. Rebuild the solution.

## Using the IronRuby .NET Extension in IronRuby

After all the hard work, we can now use our extension in IronRuby. Open the command prompt and navigate to the directory where our assembly is located (for example, `C:\SampleExtension\SampleExtension\bin\Debug`).



Run `ir.exe` to start the IronRuby REPL console. The first task now is to load the extension. This is done with the `load_assembly` method:

```
>>> load_assembly "SampleExtension", "MyRubyExtension"
true
```

The extension is loaded and we can use it. The next sample is a REPL console session that uses the IronRuby .NET extension we've just created:

```
>>> t = TextFormatter.new("IronRuby Extensions Rock!")
=> MyRubyExtension.TextFormatter
>>> t.congratulate
Congratulations!
IronRuby Extensions Rock!
=> nil
>>> t.warn!
Beware!
IronRuby Extensions Rock!
=> nil
>>> t = TextFormatter.new("IronRuby Extensions Rock!", false)
=> MyRubyExtension.TextFormatter
>>> t.congratulate
Congratulations!
IronRuby Extensions Rock!
=> nil
```

Go ahead and try it out yourself!

## Summary

In this chapter, you learned all about IronRuby .NET extensions. You learned about the process of writing an IronRuby .NET extension (and best practices for doing so), you saw different attributes of IronRuby extensions like `RubyModuleAttribute` and `RubyClassAttribute`, and you even wrote an entire IronRuby .NET extension (from creating its Visual Studio project to using it inside IronRuby code).

IronRuby .NET extensions provide developers a great and easy way to customize their .NET code to fit perfectly with IronRuby. Nonetheless, the decision whether to write an extension or to rely on IronRuby's interoperability capabilities should be considered seriously, because writing .NET extensions, even though doing so is not a complicated process, is much more complicated than just using .NET classes.

# Index

## SYMBOLS

>, >= (greater than/greater than or equal to) operator, 65  
<, <= (less than/less than or equal to) operator, 65 []  
  (array access) operator, 112, 114  
|| (Boolean OR) operator, 65  
== (equality) operator, 112  
<=> (general comparison) operator, 65  
< <= => > (order comparison) operator, 112  
() (parentheses), 45  
<< (shift-left) operator, 112  
>> (shift-right) operator, 112  
-@ (unary minus) operator, 112  
+@ (unary plus) operator, 112  
| (vertical bars), 96  
! (no pattern match) operator, 65  
!= (not equal to) operator, 65  
\$LOAD\_PATH variable, 210  
&& (Boolean AND) operator, 65  
; (semicolon), 44  
=== (case equality) operator, 65, 112, 113  
== (equal to) operator, 65  
@ (at sign), 453  
[] = (array access setter) operator, 112, 114

## A

**abbrev library**, 132  
  references, 135  
**About Your Application's Environment link**, 337  
**abstract classes**, 123, 242  
  implementation, 247  
**abstract methods**, 246-247  
**accept\_verbs method**, 374  
**accessing**. *See also* security  
  arrays, 55-56  
  CAS, 19  
**data**. *See* data access  
  file properties, 173-174  
  hashes, 58  
  HTML, Silverlight, 414-415  
  strings, 53-54  
  variables from outside, 106-107  
  XAML elements, 412-414  
**accessors**  
  classes, 107-109  
  implementation, 180  
  properties, overriding, 252  
**ActionController**, 341  
**ActionExecutionContext object**, 388

**actions**

- canceling, 388
- controllers, return values, 371
- customizing, 395-396
- filters, 387-390

**ActionView, 341****ActiveRecord model, 340-341****Ada, 1, 5****add\_EventName method, 253****adding**

- build configurations, 502-504
- code, Silverlight, 411-415
- connection strings to classes, 262
- functionality, 353-354
- IronRubyMvs Dll files, Visual Studio, 365
- layouts, 351-352
- log entries, 143
- references to assemblies, 367
- Refresh button, 360-361
- stylesheets, 350-351
- video, 418
- web pages to Silverlight, 406-408
- WinForms
  - controls, 289-293
  - functionality, 293-295

**add\_log method, 456****add\_one method, 88****Advanced System Settings, 27****AdventureWorksLT, 260****after\_action filter, 389****after method, 443****after\_result filter, 391****alias\_action method, 375****aliasing**

- methods, 499
- namespaces, 214
- syntax, 215

**alias keyword, methods, 91-92****and, 30****And directive, 448****AND operator, 65****animation**

- Silverlight, 417-418
- WPF, 324-325

**app/controllers folder, 333****App folder, 333****app/helpers folder, 334****Application class, 304****Application.run method, 305****applications**

- ASP.NET MVC, building, 365-367
- code-containing structures, 86
- controllers, creating, 375-377
- C#/VB.Net, 459-461

**data access. See data access**

- execution code, writing, 300
- Hello World!, 48
- Java, 35
- layouts, applying, 360
- models, creating, 368-371
- NetBeans, 35-36

**reflection. See reflection**

- RoR, creating, 332-337
- Ruby in Steel, 34-35
- Silverlight, 402-406, 405
- structures, WinForms, 282
- threads, 161-169
- views, creating, 382-385

**applying**

- built-in mixins, 488
- CachedDataAccess class, 278-279
- designer, Visual Studio, 295-296
- extensions
  - IronRuby, 501
  - .NET, 509-510
- gems, 183-184
- layouts, 360
- libraries, MVC, 375
- methods as block arguments, 94
- mixins, 254-255
- objects
  - IronRuby, 470-471
  - .NET, 214-231
- previous layout lists, 358-360
- regular expressions, 61
- SqlServerAccessor class, 265
- standard libraries, 131
- symbols, 58

**app/models folder, 334****app/view folder, 334****app/view/layouts folder, 334****architecture**

- DLR, 20-21, 22-23. *See also* DLR
- .NET Framework, 15-16
- REST, 339-340

**arithmetic operators, 49, 112****around\_action filter, 382****around\_result filter, 392****Array class methods, 56-57****arrays**

- accessing, 55-56
- defining, 54-55
- ranges, converting, 59
- Ruby, 54-57

**ASP.NET MVC, 363**

- applications, building, 365-367
- classic ASP.NET, 398
- environments, preparing, 363-365
- features, 398
- filters, 387-396
- installing, 364
- routes, 385-387
- validations, 396-398

**assemblies, 18**

- deleting, 366
- GAC, 18
- loading, 260, 267
- .NET
  - loading, 207-210
  - WinForms, 296
- references, adding, 367
- requirements, Chat class, 282

- running, 16
- WinForms, loading, 285
- assertions, unit testing, 428-431
- associating methods with objects, 94-95
- at sign (@), 453
- attributes
  - Window, 309-310
  - WindowState, 310-311
- audio, 418
- AuthorizationContext object, 392-393
- authorization filters, 392-393
- automatic log rotation, 144
- availability
  - libraries, 11, 133-135
  - socket services, 149-152

## B

- Background directive, Cucumber, 452
- Base Class Library. *See* BCL
- base64 library, 132
  - references, 135
- BasicSocket class, 155
- BCL (Base Class Library), 19
- BDD (behavior-driven development), 435
- be\_an\_instance\_of RSpec expression matcher, 440
- be\_an RSpec expression matcher, 440
- be\_a RSpec expression matcher, 440
- be\_close RSpec expression matcher, 440
- be\_false RSpec expression matcher, 440
- before\_action filter, 388
- before method, 443
- before\_result filter, 390-391
- BEGIN class, 77
- begin clause, 88
- behavior
  - RSpec, creating, 438
  - rules, Cucumber, 443-457
- behavior-driven development (BDD), 435
- Behavior object, 438
- be\_instance\_of RSpec expression matcher, 440
- be\_kind\_of RSpec expression matcher, 440
- benchmark library, 132, 136
- be\_nil RSpec expression matcher, 440
- be\_predicate RSpec expression matcher, 440
- best practices, extension development, 481
- be\_true RSpec expression matcher, 440
- BigDecimal library, 132
  - references, 136
- binary marshaling, 181
- binders, 24
- binding
  - data
    - Silverlight, 419-422
    - WPF, 325-329
  - dynamic data, 327-328
  - private binding mode, 213-214
  - static data, 325-326

- Bin folder, 30
- bitwise operators, 112
- block arguments, applying methods as, 94
- BlockParam parameter, 492
- blocks, 96-97, 496
  - flow, 100-101
- body parts, web pages, 352
- Booleans, 60
- break keyword, 74
- brushes
  - Silverlight, 415
  - WPF, 322-324
- BuildConfig property, 483, 485
- build configurations, adding, 502-504
- builder pattern, 196-199
- building
  - applications, ASPNet MVC, 365-367
  - chat, WinForms, 285-299
  - Chat class, 282-285
  - ChatForm class, 285-286
  - class structures, 260, 267
  - connection strings, 261, 267
  - extensions, 501-510
- built-in mixins, 488
- buttons
  - Environment Variables, 28
  - Refresh, adding, 360-361

## C

- CachedDataAccess class, 276-279
  - applying, 278-279
- cached\_data\_access.rb file, 277-278
- caches, 22
  - GAC, 18
- calc\_numerological\_value.feature, 448
- calling
  - accessors, 108
  - class methods, 111
  - methods, 45-46, 94
  - procs, 97
- call sites, 23
- CallSiteStorage parameter, 492
- canceling actions, 388
- canvas, formatting, 320-321
- Canvas control, 411
- CAS (Code Access Security), 19
- case statement, 67-69
- catching exceptions within methods, 88
- CGI (Common Gateway Interface), 132
- change RSpec expression matcher, 440
- characters, numeric values of, 51
- chat, building WinForms, 285-299
- Chat class, 282
  - building, 282-285
- Chatform class, 282, 297-299
  - building, 285-286

**chat.rb file, 284-285**

**ChatRunner class, 282**

**Chiron, initParams parameter, 407**

**CIL (Common Intermediate Language), 17-18**

**Class class methods, 232-233**

**classes**

- abstract, 123, 247
- accessors, 107-109
- Application, 304
- Array, methods, 56-57
- BasicSocket, 155
- BCL, 19
- BEGIN, 77
- CachedDataAccess, 276-279
- Chat, 282-285
- ChatForm, 282, 285-286
- Chatform, 297-299
- ChatRunner, 282
- Class, methods, 232-233
- CLR, 216, 217
  - inheritance from, 239-243
  - opening, 254-256
- code standards, 47
- connection strings, adding, 262
- constants, 105-106
- defining, 93
- duck typing, 124-126
- END, 77
- errors, customizing, 85-86
- Exception, methods, 78
- ExceptionContext, 393-394
- exceptions, 490-491
- extensions, 488-491
- File, 170
- Form, initializing, 286
- generic, 241-242
- Hash, methods, 58-59
- helper, 349-350
- inheritance, 120-124
- instances, creating, 102
- IPSocket, 156
- IronRuby, 235-236
- IronRuby::Clr, 236
- methods, 109-111
  - overriding, 122
  - undefining, 491
- module-contained objects, 126
- modules, 126
- Mutex, 167-168
- MySQLAccessor, 272
- Numerology.Calculator, 426
- Object
  - methods, 231-232
  - opening, 255-256
- Recorder, implementation, 38-39
- regular, 239-242
- Ruby, 101-126
- RubyClassAttribute properties, 488-489
- RubyMethodAttributes, 492, 494-495
- ScriptEngine, 23, 463-465
- ScriptRuntime, 23, 462-463

ScriptRuntimeSetup, 462

ScriptScope, 23, 465-466

ScriptSource, 23, 466-467

sealed, 243

singleton, 490

SqlServerAccessor, applying, 265

Stack, 38

static, 243

String, 53, 54, 234-235

structures, building, 260, 267

System.String, 254

System.Windows.Forms.Application, 300

TCPServer, 156

TCPSocket, 156, 283

ToDoListModel, 370

UDPSocket, 156

variables, 102-107, 103

visibility control, 118-120

**classic ASP.NET, 398**

**class keyword, 101**

**clauses**

begin, 88

block, 96

ensure, 82-83

rescue statement, 79

**CLI (Common Language Infrastructure), 14-18**

**ClientSize property, 287**

**closures, blocks, 97. See also blocks**

**CLR (Common Language Runtime), 1, 15, 17**

classes, 216, 217

inheritance from, 239-243

opening, 254-256

constants, 222

delegates, 217-218

fields, 228

interfaces, 216, 243-244

members, hiding, 487

namespaces, converting, 214

naming conventions, 212-213

objects

applying Recorder class on, 38

reflection, 237

properties, 228-229

Ruby, type differences, 211

structs, inheritance from, 243

**clr\_constructor method, 232**

**clr\_ctor method, 232**

**clr\_member method, 231**

**clr\_members method, 233**

**clr\_new method, 233**

**CoC (Convention over Configuration), 340**

**code**

compiling, 467

creating, 504-506

dynamically, executing, 180-181

execution, 300-301, 467

file structure, 46-47

Gherkin, 451

hosts, 26

naming, 212

- .NET
  - Framework, 16
  - mapping, 210-214
  - standards, 211-213
- reading, 467
- reflection. See reflection**
- RSpec, injecting, 442-444
- Ruby, 2-5. *See also* Ruby
- ScriptEngine class, executing, 464-465
- ScriptRuntime class, executing, 463
- Silverlight, adding, 411-415
- source. See source code**
- standards, 47
- unit testing, 426-427
- XAML, 305-307
- Code Access Security (CAS), 19**
- code-containing structures, Ruby, 86**
- CodePlex, 22, 26**
- Collatz conjecture, 77**
- collisions**
  - inheritance, 124
  - .NET, mapping code, 210-214
- command-line**
  - arguments for file execution mode, 33-34
  - chr tool, 404-406
  - databases, creating from, 337
  - sl tool, 402-403
  - switches, 31
- commands**
  - db:migrate, 345-346
  - icucumber, 457
  - patterns, 190-192
  - scaffold, 344, 345
  - script/destroy, 345
  - script/generate, 343-345
  - script/generate controller, 346
  - script/server, 342
- comma-separated value. See CSV**
- comments**
  - code standards, 47
  - syntax, 43-44
- Common Gateway Interface. See CGI**
- Common Intermediate Language. See CIL**
- Common Language Infrastructure. See CLI**
- Common Language Runtime. See CLR**
- Common Type System (CTS), 16**
- comparison operators, 64-65, 112, 114**
- Compatibility property, 483, 485**
- compilers, DLR, 24**
- compiling**
  - code, 467
  - JIT, 16
- complex library, 132**
  - references, 137
- components**
  - RoR, 340-342
  - runtime, 22, 23-24
- conditions, 64-74**
- Config folder, 334**
- configuration method, 235**
- configuring**
  - applications, RoR, 332-337
  - behavior, RSpec, 438
  - build, adding, 502-504
  - class instances, 102
  - controllers, 346-349
  - databases, RoR, 334-337
  - extensions, .NET, 478-501
  - form properties, 287-289
  - keys for SQL Server connections, 261
  - PATH Environment Variable, 29
  - RubyMine, 37
  - ScriptRuntime class, 462-463
  - values to variables, 44-45
  - views, 346-349
  - visibility control, 118-120
  - web pages, 346-354
- connections**
  - MySQL, opening, 268
  - SQL Server, opening, 262
  - strings
    - adding, 262
    - building, 261, 267
    - examples of, 261
- connectors, MySQL, 260**
- consoles, 11**
  - keys, 408
  - modes, REPL, 31
- constants**
  - classes, 105-106
  - CLR, 222
  - code standards, 47
  - extensions, 501
  - mapping, 222
  - module-contained objects, 127
  - Ruby, 63-64
- const-missing method, 118**
- constructors**
  - defining, 102
  - inheritance, 241
  - methods, 497
- contacting**
  - MySQL, 265-272
  - SQL Server, 260-265
- content, WPF, 315-317**
- contribution pages, 26**
- ControlBox property, 287**
- controllers**
  - ActionController, 341
  - actions, return values, 371
  - applications, creating, 375-377
  - ASP.NET MVC validations, 396-397
  - creating, 343, 346-349
  - features, 371-372
  - filters, 394
  - MVC, 371-372
  - views, creating, 378

**controls**

## layouts

Silverlight, 410-411

WPF, 317-321

Silverlight, 411

StackPanel, 317-319

structures, Ruby, 64-77

WinForms, adding, 289-293

WrapPanel, 318-319

**Convention over Configuration. See CoC****conventions, naming, 212**

Ruby, 484

unit testing, 427-428

**converting**

namespaces, CLR, 214

ranges to arrays, 59

**CRUD (Create Read Update Delete), 259, 341****CSV (comma-separated value), 132**

references, 137

**CTS (Common Type System), 16****Cucumber, 443-457**

Background directive, 452

executing, 457

features, 446-447

hooks, 454-455

installing, 445

multilanguage, 456-457

project structures, 445-446

scenarios, 447-452

tags, 453-454

worlds, 456

**culture of assemblies, 18****customizing**

error classes, 85-86

filters, 395-396

index pages, 356

installation, 26

routes, 386-387

**C#/VB.Net, 458-459**

applications, 459-461

external libraries, 472-473

IronRuby, executing from, 468-473

ScriptEngine class, 463-465

ScriptRuntime class, 462-463

ScriptScope class, 465-466

ScriptSource class, 466-467

**D****-D, 33****-d, 33****data access, 256**

environments, preparing, 260-259

overview, 259

SQL Server, contacting, 260-265

**databases. See also MySQL; SQL Server**

MySQL, preparing, 266

queries, 263-264, 268

RoR configuration, 334-337

web pages, formatting, 354-361

**data binding**

Silverlight, 419-422

WPF, 325-329

**data templates, Silverlight, 422****dates**

and times, Ruby, 62-63

libraries, 132

**DayError, 86****Db folder, 334****db:migrate command, 345-346****debug key, 408****Debug library, 132****DebugMode property, 462****declaring**

floats, 49

integers, 49

types, 48

**default parameter values, 92****Defineln property, 483, 486****defining**

arrays, 54-55

blocks, 96

classes, 93

class methods, 110

constants, 105

constructors, 102

exception types, 139

hashes, 57

Lambdas, 99

methods, 88-89, 95-96

operator behavior, 111

proc objects, 97

regular expressions, 60-61

strings, 50

types, 44

**def keyword, 88****Delegate library, 132****delegates, CLR, 217-218****deleting**

assemblies, 366

method definitions, 95-96

records, MySQL, 269

resources, 345

**delimiters, strings, 50****describe method, 438****design**

MySQL, 272-276

patterns, 186-202

builder, 196-199

command, 190-192

iterator, 188-190

observer, 194-196

singleton, 192-194

strategy, 186-188

SQL Server, 272-276

**designer, Visual Studio, 295-296****development**

environments, 34-37

**integrated development environments. See IDEs****dialog boxes**

Edit System Variable, 28

New Project, 366

differences between Lambdas and Procs, 99

digest library, 132

references, 138

directives

And, 448

Background, Cucumber, 452

directories

listing, 174-175

structures, 333-334

templates, sl tool, 403

Distributed Ruby. *See* Drb

dividing integers, 50

DLR (Dynamic Language Runtime), 1

architecture, 22-23

features, 23-24

overview of, 20-21

Doc folder, 334

documents. *See also* text

here, 51

words, modifying, 357

XAML, 305-307

XML

generating, 153

reading, 154

domain-specific languages. *See* DSLs

Don't Repeat Yourself. *See* DRY

DoubleAnimation element, 324

double-quoted strings, 44-50

downloading

code, 26

IronRubyMvs Dll files, 364-365

.NET Framework, 26

standard libraries, 159

Drb (Distributed Ruby), 132

DRY (Don't Repeat Yourself), 340

DSLs (domain-specific languages), 199-202

duck typing, 8-9

classes, 124-126

dynamic data

binding to, 327-328

Silverlight, 420-421

Dynamic Language Runtime. *See* DLR

dynamic languages, 6-7, 20-21

implementation, 24

dynamic messages, 139

## E

each method, 59, 116

-e "command," 33

Edit System Variable dialog box, 28

Eiffel, 1, 5

elements

DoubleAnimation, 324

MediaElement, 418

Silverlight, retrieving, 412-414

Storyboard, 418

TextBlock, 304

WPF, retrieving, 308

else statements, 81-82

e2mmap library, 132, 139

empty namespaces, 214

END class, 77

end keyword, 88

English library, 132

references, 140-141

ensure clause, 82-83

entries, adding log, 143

enumerable objects, 72-73

enums, 221-222

environments

ASP.NET MVC, preparing, 363-365

data access, preparing, 260-259

development, 34-37

integrated development environments. *See* IDEs

reflection. *See* reflection

RoR

navigating, 342-346

preparing, 331-332

Silverlight, preparing, 402

target, .NET extensions, 482

Environment Variables button, 28

env.rb file, 446

Eql RSpec expression matcher, 440

Equal RSpec expression matcher, 441

Erb library, 132

references, 141-143

Eregexp library, 132

error.js file, 408

errors

classes, customizing, 85-86

DayError, 86

IOError, 80

StandardError, 85

SyntaxError, 80

events

handling, Silverlight, 414

layout, suppressing, 286

.NET, 218-221

overriding, 253-254

startup, 305

subscribing, 219-220

TextBlock.Loaded, 418

unsubscribing, 220

WPF, handling, 308-309

examples, tables, 451-452

Exception class methods, 78

ExceptionContext class, 393-394

exceptionDetail key, 408

exceptions

catching within methods, 88

classes, 490-491

filters, 393-394

handling, 78-86

raising, 83-85



threads, 163  
types, defining, 139

**executables, 30-34****executing**

code, 467  
dynamically, 180-181  
RSpec, 443  
ScriptEngine class, 464-465  
ScriptRuntime class, 463  
Cucumber, 457  
IronRuby from C#/VB.Net, 468-473  
virtual methods, 246

**execution code, writing, 300-301****expectation methods**

RSpec, 439-442

**expressions**

interpolation, 52  
matchers, RSpec, 441-442  
regular, 60-62  
trees, 23, 24

**extending IronRuby, 473-478****Extends property, 484, 486****eXtensible Application Markup Language. See XAML****extensions**

classes, 488-491  
constants, 501  
infrastructure, 478-481  
IronRuby  
applying, 501  
building, 501-510  
methods, 491-500  
.NET  
applying, 509-510  
creating, 478-501  
modules, 482-488  
projects, 481  
Visual Studio, creating projects, 502

**external libraries, C#/VB.Net, 472-473****F****features**

ASP.NET MVC, 398  
controllers, 371-372  
Cucumber, 446-447  
DLR, 23-24  
installation, 26  
.NET Framework, 16-20  
Ruby, 6-11  
tagging, 453

**fields, CLR, 228****File class, 170****filenames, loading .NET assemblies, 208****File.open method, 170****files, 169-175**

cached\_data\_access.rb, 277-278  
chat.rb, 284-285  
code structure, 46-47  
env.rb, 446  
error.js, 408

initial project, creating, 333  
IronRuby, executing, 468  
IronRubyMvs Dll, adding to Visual Studio, 365  
mysql.rb, 270-272  
operations, 175  
properties, accessing, 173-174  
reading, 158, 170-172  
sql.rb, 264-265  
writing, 158, 172-173  
ZIP, 27

**FileUtils library, 132**

references, 143

**filters**

actions, 387-390  
ASP.NET MVC, 387-396  
authorization, 392-393  
controllers, 394  
customizing, 395-396  
exceptions, 393-394  
results, 390-392

**finding**

gems, 185  
living objects, 176-177  
standard libraries, 159

**Find library, 133****find\_name method, 308****flags, regular expressions, 61****floats, declaring, 49****flow**

filters, 387  
keywords, 100-101  
loops, modifying, 74

**folders, 29-30**

Lib, 209  
RoR, 333-334  
structures, sl tool, 403

**for loops, 71****formatting**

applications  
controllers, 375-377  
RoR, 332-337  
views, 382-385  
behavior, RSpec, 438  
canvas, 320-321  
class instances, 102  
code, 504-506  
controllers, 346-349  
databases, web pages, 354-361  
extensions, .NET, 478-501  
form properties, 287-289  
MVC views, 378-385  
views, 346-349  
visibility control, 118-120  
web pages, 346-354

**FormBorderStyle property, 287****Form class, initializing, 286****forms**

data, posting, 145  
properties, formatting, 287-289  
WinForms, 280. *See also* WinForms

**Forwardable library, 133**

**frameworks**

- Cucumber, 443-457
- .NET
  - Framework, 20
  - overview of, 11-13

**RoR. See RoR**

- RSpec, 435-444

**WPF. See WPF**

full paths, loading .NET assemblies, 208

**functionality**

- adding, 353-354
- WinForms, adding, 293-295

**G**

**GAC (Global Assembly Cache), 18**

**Garbage Collector, 176**

**gems**

- applying, 183-184
- finding, 185
- installing, 183
- RoR, installing, 332

**generating**

- ASP.NET MVC initial projects, 365-367
- helper classes, 344
- resources, web pages, 354-355
- XML documents, 153

**Generator library, 133**

**generic classes, 229-231, 241-242**

**GetOptLong library, 133**

**get\_products method, 276**

**Gherkin code, 451**

**GitHub, 26**

**Given step, 447**

**Global Assembly Cache. See GAC**

**global hooks, 454**

**globals method, 235**

**graphics**

- Silverlight, 415-417
- WPF, 321-325

**Grid control, 410**

**grids, 319-320**

**GServer library, 133**

**guidelines, RoR, 339-311**

**GUI (graphical user interface) tools, installing, 260**

**H**

**-h, 34**

**Hailstone Sequence, 77**

**handled property, 309**

**handling**

- events
  - Silverlight, 414
  - WPF, 308-309
- exceptions, 78-86
- files, 169-175

**Hansson, David Heinemeier, 1**

**Hash class methods, 58-59**

**hashes, 57-59**

- accessing, 58
- defining, 57

**have\_at\_least RSpec expression matcher, 441**

**have\_at\_most RSpec expression matcher, 441**

**Have RSpec expression matcher, 441**

**head parts, web pages, 352**

**Hello World!, 48**

**helper classes, 349-350**

- generating, 344

**helper methods**

- HTML, 378-379
- views, 380

**here documents, 51**

**HideClrMembers property, 484**

**hiding**

- CLR members, 487
- methods, 500

**highlighting syntax, 35**

**history**

- of .NET Framework, 13-14
- of Ruby, 5-6

**hooks, Cucumber, 454-455**

**HostArguments property, 462**

**hosts**

- code, 26
- models, 22, 23

**HostType property, 463**

**HTML (Hypertext Markup Language)**

- helper methods, 378-379
- Silverlight, accessing, 414-415

**I**

**-I "command," 33**

**icucumber command, 457**

**identifiers, tags, 453**

**IDEs (integrated development environments), 35**

**if, 65-67**

**if-else statements, 69**

**igem.bat, 30**

**iir.bat, 30**

**iirb.bat, 30**

**implementation, 24-25**

- abstract classes, 247
- accessors, 180
- dynamic languages, 24
- invoking superclass method, 123
- Recorder class, 38-39
- regular methods, 247-248
- Ruby, 6
- sealed methods, 250-251
- static methods, 248-249
- steps, 449-451
- threads, 162
- virtual methods, 245-246

- include method, 215
- include RSpec expression matcher, 441
- indexer methods, 224-225
- index pages, customizing, 356
- infrastructure extensions, 478-481
- inheritance
  - classes, 120-124
  - CLR classes, 239-243
  - CLR interfaces, 243-244
  - CLR structs, 243
  - collisions, 124
  - constructors, 241
  - numeric types, 49
- Inherits parameter, 489-490
- Initialize method, 102, 498-499
- initializers, libraries, 508-509
- initializing
  - Form class, 286
  - ScriptRuntime class, 462-463
  - visibility, 102
- initial project files, creating, 333
- initiating Chat class, 282-283
- initParams parameter, 407
- injecting code, RSpec, 442-444
- inserting records, MySQL, 269
- installing, 26-30
  - ASP.NET MVC, 364
  - Cucumber, 445
  - features, 26
  - gems, 183, 332
  - GUI tools, 260
  - RSpec, 436
  - SQL Server, 332
- instances
  - classes, creating, 102
  - methods, 222
  - variables, 103-104, 453
- integers
  - declaring, 49
  - dividing, 50
- integrated development environment. *See* IDEs
- integration, 1
- intellisense, 35
- interfaces
  - CGI, 132
  - CLR, 216, 243-244
  - programming, 506-508
- WinForms. *See* WinForms
- interoperability
  - .NET, 203
- interpolation
  - expressions, 52
- interpreters, 31. *See also* ir.exe
  - RubyMine, 37
- inverted until loops, 71
- inverted while loops, 70
- investigating objects, 177-178
- invoking
  - methods, 178-180
  - superclass method implementation, 123
- IOError, 80

- IPAddr library, 133
- IPSocket class, 156
- irackup.bat, 30
- irails.bat, 30
- irake.bat, 30
- irake tool, 337
- irdoc.bat, 30
- ir.exe, 30
  - command-line arguments for file execution mode, 33-34
- ir64.exe, 30
- IronRuby, 6. *See also* Ruby
  - classes, 235-236
  - C#/VB.Net, 468-473
  - extending, 473-478
  - extensions
    - applying, 501
    - building, 501-510
  - objects, applying, 470-471
- IronRuby::Clr class, 236
- IronRubyMvs DLL files
  - adding, 365
  - downloading, 364-365
- iterator pattern, 188-190

## J

- Java applications, 35
- Java Virtual Machine. *See* JVM
- JCode library, 133
- JIT (just-in-time) compiling, 16
- join method, 162
- JRuby, 6
- just-in-time. *See* JIT
- JVM (Java Virtual Machine), 14

## K

- Kconv library, 133
- keys, initParams Chiron-related, 408
- keywords, flow, 100-101
- K[kcode], 33

## L

- lambdas, 99-100, 496
  - flow, 100-101
- Language-Integrated Query. *See* LINQ
- languages, 1
  - context, 24
  - DSLs, 199-202
  - dynamic, 6-7, 20-21
  - object-oriented, 7
  - programming, 13
- Ruby. *See* Ruby
- LanguageSetups property, 463
- layout
  - controls, WPF, 317-321
  - events, suppressing, 286

**layouts.** *See also* **formatting**  
 adding, 351-352  
 applications, applying, 360  
 previous layout lists, applying, 358-360  
 Silverlight, 410-411

**Lib folder, 209, 334**

**libraries**  
 available, 11  
 BCL, 19  
 code without IronRuby extension methods, 505  
 DLR, 22. *See also* DLR  
 external, C#/VB.Net, 472-473  
 initializers, 508-509  
 MVC, 375  
 RSpec requirements, 436-437  
 standard, 130-131. *See also* **standard libraries**

**Libs folder, 30**

**licenses, MS-PL, 22**

**limitations of RubyGems, 185**

**LINQ (Language-Integrated Query), 279-280**

**Linux operating system, 14**

**Lisp, 5**

**listen method, 283**

**listing directories, 174-175**

**lists, applying previous, 358-360**

**living objects, finding, 176-177**

**load\_assembly method, 209, 501**

**load\_component method, 412**

**loaded\_assemblies method, 235**

**loaded\_scripts method, 236**

**loading assemblies, 260, 267**  
 .NET, 207-210  
 WinForms, 285

**load method, 210, 236**

**load\_root\_visual method, 412**

**local variables, 105**

**log entries, adding, 143**

**Log folder, 334**

**logger library, 133**  
 references, 143

**loops, 70-72**  
 flow, modifying, 74  
 for, 71  
 inverted until, 71  
 inverted while, 70  
 loop loops, 72  
 numbers, 73-74  
 ranges, 74  
 until, 70  
 while, 70

## M

**MacRuby, 6**

**MailRead library, 133**

**managing memory, .NET Framework, 19**

**manual installation, 27. *See also* installation**

**mapping**  
 constants, 222  
 namespaces, 256  
 .NET code, 210-214  
 objects, 211  
 ORM, 341

**marshaling, 181-182**  
 binary, 181  
 textual, 182

**matchers, expressions, 441-442**

**Match RSpec expression matcher, 441**

**MathN library, 133**

**Matrix library, 133**

**Matsumoto, Yukihiro, 1**

**Matz's Ruby Interpreter. *See* MRI**

**MaximizeBox property, 287**

**media. *See also* animation; graphics**  
 Silverlight, 417-418

**MediaElement element, 418**

**members, hiding CLR, 487**

**memory management, .NET Framework, 19**

**messages**  
 dynamic, 139  
 receiving, 283  
 sending, 283

**metadata, 16**

**metaprogramming, 9-10**

**method\_missing method, 38, 117**

**methods, 222-227**  
 abstract, 246-247  
 accept\_verbs, 374  
 add\_EventName, 253  
 add\_log, 456  
 add\_one, 88  
 after, 443  
 alias\_action, 375  
 aliasing, 499  
 alias keyword, 91-92  
 Application.run, 305  
 Array class, 56-57  
 assert, unit testing, 428-431  
 before, 443  
 calling, 45-46  
 Class class, 232-233  
 classes, 109-111  
   overriding, 122  
   undefining, 491  
 clr\_constructor, 232  
 clr\_ctor, 232  
 clr\_member, 231  
 clr\_members, 233  
 clr\_new, 233  
 code standards, 47  
 configuration, 235  
 const-missing, 118  
 constructors, 497  
 defining, 88-89, 95-96  
 describe, 438

- each, 59, 116
- Exception class, 78
- expectation, RSpec, 439-442
- extensions, 491-500
- File.open, 170
- find\_name, 308
- get\_products, 276
- globals, 235
- Hash class, 58-59
- helper. *See* helper methods
- hiding, 500
- include, 215
- indexers, 224-225
- Initialize, 498-499
- initialize, 102
- instances, 222
- invoking, 178-180
- IronRuby class, 235-236
- join, 162
- listen, 283
- load, 236
- load\_assembly, 501
- load\_component, 412
- loaded\_assemblies, 235
- loaded\_scripts, 236
- load\_root\_visual, 412
- method-missing, 117
- method\_missing, 38
- Module, 108
- module-contained objects, 127
- multiple overloads, 249-250
- multiply, 254
- naming, 90
- NavigationWindow, 314-315
- new, 161
- non\_action, 374
- Object class, 231-232
- objects, associating, 94-95
- open, 171
- overload, 234
- overloaded, 223
- overriding, 121-122, 245-251
- overview of, 87-88
- parameters, 495
- parse, 307
- print, 48
- printf, 52-53
- private, 223
  - overriding, 122
  - testing, 428
- public, 222
- regular, 247-248
- remove\_EventName, 253
- require, 131, 207, 236
- run, 169, 300
- sealed, 250-251
- selectors, 374
- setter, 117
- setup, unit testing, 432-433
- singleton, 96

- special, 115-118
- special IronRuby, 231-236
- special .NET, 225
- special parameters, 493
- start, 161
- static, 223, 248-249
- String class, 54, 234-235
- String.new, 51
- succ, 116
- teardown, unit testing, 432-433
- the\_next\_big\_thing, 92
- to\_clr\_type, 231
- undef, 96
- values, returning from, 90-91
- virtual, 245-246
- visibility control, 118-120
- Microsoft Intermediate Language (MSIL).** *See* CIL
- Microsoft Public License (MS-PL),** 22
- Microsoft SQL Server.** *See* SQL Server
- MinimizeBox property,** 287
- mixins**
  - applying, 254-255
  - built-in, 488
  - definitions, 487-488
  - modules, 128-129
- models**
  - ActiveRecord, 340-341
  - ASP.NET MVC validations, 396
  - creating, 343
  - hosts, 22, 23
  - MVC, 368-371
- model-view-controller.** *See* MVC
- modes**
  - consoles, REPL, 31
  - private binding, 213-214
- modifier statements,** 80
- modifying**
  - databases, 335
  - loop flow, 74
  - print separators, 172
  - visibility control, 118-120
  - words, 357
- Module method,** 108
- ModuleRestrictions enum values,** 486-487
- modules, 126-129**
  - classes, 126
  - code standards, 47
  - mixins, 128-129
  - module-contained objects, 126-127
  - namespaces, 127-128, 256
  - .NET extensions, 482-488
- monitor library,** 133
  - references, 144
- monitors, 168-169**
- Mono,** 17
- MRI (Matz's Ruby Interpreter),** 6
- MSIL (Microsoft Intermediate Language).** *See* CIL
- MS-PL (Microsoft Public License),** 22
- multilanguage, Cucumber,** 456-457
- multiline comments,** 43

multiple overloads, methods with, 249-250  
 multiple-parameter replacement, 89  
 multiple statements, writing, 44  
 multiply method, 254  
 Mutex class, 167-168  
 Mutex\_m library, 133  
 MVC (model-view-controller), 339, 368-385  
 ASP.NET MVC. *See* ASP.NET MVC  
   controllers, 371-372  
   models, 368-371  
   views, 378-385  
 My Computer, 27  
 MySQL, 260  
   connections, opening, 268  
   connectors, 260  
   contacting, 265-272  
   databases, preparing, 266  
   design, 272-276  
   records  
     deleting, 269  
     inserting, 269  
 MySQLAccessor class, 272  
 mysql.rb files, 270-272

## N

Name property, 483, 484  
 namespaces  
   aliasing, 214  
   mapping, 256  
   modules, 127-128  
   .NET objects, 214-215  
   opening, 256  
   XAML, 306  
 naming  
   assemblies, 18  
   code, 212  
   conventions  
     Ruby, 484  
     unit testing, 427-428  
   methods, 90  
 navigating environments, RoR, 342-346  
 NavigationWindow, 314-315  
 nested test suites, 434  
 .NET, 1  
   assemblies  
     loading, 207-210  
     WinForms, 296  
   code  
     mapping, 210-214  
     standards, 211-213  
   events, 218-221  
   extensions  
     applying, 509-510  
     creating, 478-501  
     modules, 482-488

Framework  
   downloading, 26  
   features, 16-20  
   frameworks, 20  
   history of, 13-14  
   memory management, 19  
   overview of, 11-13, 15-16  
   security, 19  
   versions, 14  
   interoperability, 203  
   objects, applying, 214-231  
   special methods, 225  
 NetBeans, 35-36  
 Net/ftp library, 133  
 Net/ftplib library, 133  
 Net/http library, 133  
   references, 144  
 Net/imap library, 133  
 Net/pop library, 133  
 Net/smtp library, 133  
 Net/telnet library, 133  
 Net/telnet library, 133  
 new method, 161  
 New Project dialog box, 366  
 Next Generation Windows Services. *See* NGWS  
 next keyword, 75  
 NGWS (Next Generation Windows Services), 14  
 non\_action method, 374  
 Notepad++, 37  
 Novell, 14  
 numbers  
   loops, 73-74  
   Ruby, 48-50  
 Numerology.Calculator class, 426

## O

Object class  
   methods, 231-232  
   opening, 255-256  
 object-oriented languages, 7  
 Object Rational Mapping. *See* ORM  
 objects  
   ActionExecutingContext, 388  
   AuthorizationContext, 392-393  
   Behavior, 438  
   CLR  
     applying Recorder class on, 38  
     reflection, 237  
   enumerable, 72-73  
   investigating, 177-178  
   IronRuby, applying, 470-471  
   living, finding, 176-177  
   mapping, 211  
   methods, associating, 94-95  
   module-contained, 126-127

- .NET, applying, 214-231
  - procs, defining, 97
  - receiving exception, 79
  - Ruby, 237
- ObjectSpace module, 176**
- Observer library, 133**
  - references, 145
- observer patterns, 194-196**
- onerror parameter, 407**
- opening**
  - CLR classes, 254-256
  - connections
    - MySQL, 268
    - SQL Server, 262
  - namespaces, 256
  - Object class, 255-256
- Open3 library, 134**
- open method, 171**
- open source projects, 6**
- OpenSSL library, 134**
- open-uri library references, 145**
- operating systems, 14**
- operations, files, 175**
- operators, 16**
  - AND, 65
  - arithmetic, 49, 112
  - array access ([]), 112, 114
  - array access setter ([ ] =), 112, 114
  - bitwise, 112
  - Boolean AND (&&), 65
  - Boolean OR (||), 65
  - case equality (===), 65, 112, 113
  - comparison, 64-65, 112, 114
  - equality (==), 112
  - equal to (==), 65
  - general comparison ( <=> ), 65
  - greater than/greater than or equal to ( >, >= ), 65
  - less than/less than or equal to ( <, <= ), 65
  - no pattern match (!), 65
  - not equal to (!=), 65
  - OR, 65
  - order comparison ( < <= > >= ), 112
  - overloading, 111-115
  - pattern match, 65, 112
  - precedence, 64
  - shifting, 113
  - shift-left (<<), 112
  - shift-right (>>), 112
  - ternary, 69
  - unary, 113
  - unary minus (-@), 112
  - unary plus (+@), 112
- options, installation, 26**
- Options property, 463**
- Optparse library, 134**
- ORM (Object Rational Mapping), 341**
- OR operator, 65**
- or property, 287**
- OStruct library, 134**
- outline scenarios, 451-452**
- out parameter, 226**

- overloading**
  - methods, 223
  - operators, 111-115
- overload method, 234**
- override keyword, 245**
- overriding**
  - class methods, 122
  - events, 253-254
  - methods, 121-122, 245-251
  - private methods, 122
  - properties, 251-253

## P

- pages. See web pages**
- panels, websites, 321**
- parameters**
  - default values, 92
  - Inherits, 489-490
  - initParams, 407
  - load\_assembly method, 209
  - methods, 495
  - multiple-parameter replacement, 89
  - onerror, 407
  - out, 226
  - params, 227
  - ref, 226-227
  - scope, 466
  - source, 407
  - special methods, 493
  - special types, 93-94
- params parameter, 227**
- parentheses (), 45**
- ParseDate library, 134**
- parse method, 307**
- Parse\_Tree library, 134**
- Pascal casing in .NET code, 212**
- passing**
  - data between windows, 312-314
  - data in and out of threads, 164
  - filenames, 208
  - variables to and from IronRuby, 469-470
- pass method, 165**
- PATH Environment Variable, configuring, 29**
- PathName library, 134**
- paths**
  - full, loading .NET assemblies, 208
  - searching, 464
- PATH variable, 332**
- pattern match operator, 65, 112**
- patterns, design, 186-202**
  - builder, 196-199
  - command, 190-192
  - iterator, 188-190
  - observer, 194-196
  - singleton, 192-194
  - strategy, 186-188
- Perl, 1, 5**

- ping library, 134
  - references, 147
- positioning
  - arguments with default values, 92
  - block arguments, 94
- posting form data, 145
- PP library, 134
- precedence, operators, 64
- preparing
  - environments
    - ASPNet MVC, 363-365
    - for data access, 260-259
    - RoR, 331-332
    - Silverlight, 402
  - MySQL databases, 266
- PrettyPrint library, 134
- previous layout lists, applying, 358-360
- printf method, 52-53
- print method, 48
- print method, modifying separators, 172
- priority, threads, 164-165
- private binding mode, 213-214
- PrivateBinding property, 463
- private methods, 223
  - overriding, 122
  - testing, 428
- procs, 97-99, 496
  - flow, 100-101
- profile, 33
- programming
  - interfaces, Ruby, 506-508
  - languages, 13. *See also* languages
  - metaprogramming, 9-10
- programs, Hello World!, 48
- progress\_proc, 146
- project files, creating, 333
- projects
  - extensions, 481
  - structures, Cucumber, 445-446
  - Visual Studio, creating extensions, 502
- properties
  - CLR, 228-229
  - files, accessing, 173-174
  - forms, formatting, 287-289
  - NavigationWindow, 314-315
  - overriding, 251-253
  - redefining, 252
  - RubyClassAttribute class, 488-489
  - RubyMethodAttribute class, 492
  - RubyModuleAttribute, 483-484
  - ScriptRuntimeSetup class, 463
- proxies, 146
- PStore library, 134
- Public folder, 334
- public/images folder, 334
- public/javascripts folder, 334
- public key tokens, assemblies, 18
- public methods, 222
- public/stylesheets folder, 334

## Q

- queries
  - databases, 263-264, 268
  - LINQ, 279-280
- queues, 168

## R

- Racc/parser library, 134
- RadRails, 37
- raise\_error RSpec expression matcher, 441
- raising exceptions, 83-85
- Rake, 184-185
- rakefiles, 184-185
- ranges
  - arrays, converting, 59
  - loops, 74
  - Ruby, 59-60
- rational library, 134
  - references, 152
- Readbytes library, 134
- read-evaluate-print loop. *See* REPL
- reading
  - code, 467
  - files, 158, 170-172
  - XML documents, 154
- receiving
  - exception objects, 79
  - messages, 283
- Recorder class, implementation, 38-39
- records, MySQL
  - deleting, 269
  - inserting, 269
- redefining properties, 252
- redirect results, 373-374
- redo keyword, 75
- references
  - assemblies, adding, 367
  - standard libraries, 135-158
- reflection, 176
  - CLR objects, 237
  - living objects, finding, 176-177
  - methods, invoking, 178-180
  - objects, investigating, 177-178
  - variables, configuring dynamically, 178-180
- ref parameter, 226-227
- Refresh button, adding, 360-361
- regular classes, 239-242
- regular expressions, 60-62
- regular methods, 247-248
- remove\_EventName method, 253
- removing assemblies, 366
- replacing
  - multiple-parameters, 89
  - Silverlight content, 412



**REPL (read-evaluate-print-loop), 10-11**

- console mode, 31
- WPF, 329-330

**reportErrors key, 408****Representational State Transfer. See REST****requirements**

- assemblies, Chat class, 282
- libraries, RSpec, 436-437
- .NET Framework assemblies, 208

**require method, 131, 207, 236****rescue statement, 78-80****ResizeMode values, 311-312****resources**

- deleting, 345
- panels, 321
- standard libraries, 159
- web pages, generating, 354-355

**respond\_to RSpec expression matcher, 442****REST (Representational State Transfer), 339-340****Restrictions property, 484, 486****results**

- filters, 390-392
- redirects, 373-374
- views, 372-373

**retrieving**

- pages, 144
- Silverlight elements, 412-414
- WPF elements, 308

**retry keyword, 81****return keyword, 90, 99****return values**

- controller actions, 371
- methods, 90-91
- threads, 162

**Rexml library, 134**

- references, 153

**-r "library," 33****Root folder, 30****root-visual property, 412****RoR (Ruby on Rails), 330-331**

- applications, creating, 332-337
- components, 340-342
- database configuration, 334-337
- environments
  - navigating, 342-346
  - preparing, 331-332
- folders, 333-334
- guidelines, 339-311
- servers, running, 337
- web pages
  - creating, 346-354
  - formatting database-driven pages, 354-361

**rotating automatic logs, 144****routes**

- ASP.NET MVC, 385-387
- customizing, 386-387
- RoR, 341
- URLs, 371

**RSpec, 435-444**

- behavior, creating, 438
- code, injecting, 442-444

- examples, 439

- expectation methods, 439-442

- expression matchers, 441-442

- installing, 436

- library requirements, 436-437

- running, 437

**RSS library, 134****Rubinius, 6****Ruby**

- accessors, 107-109
- arrays, 54-57
- blocks, 96-97
- classes, 101-126
- code, naming, 212
- code-containing structures, 86
- constants, 63-64
- control structures, 64-77
- dates and times, 62-63
- features, 6-11
- Hello World!, 48
- history of, 5-6
- implementation, 6
- Lambdas, 99-100
- naming conventions, 484
- .NET, loading assemblies, 207-210
- numbers, 48-50
- objects, 237
- overview of, 2-5, 25-26
- process, 97-99
- programming interfaces, 506-508
- ranges, 59-60
- regular expressions, 60-62
- symbols, 58
- syntax, 43-47
- text, 50-54
- threads, 161-169
- type differences, CLR and, 211
- variables, 48-64

**RubyClassAttribute class properties, 488-489****RubyClass parameter, 492****RubyContext parameter, 492****RubyGems, 183****gems**

- applying, 183-184
- finding, 185
- installing, 183
- installing, 183
- limitations, 185

**Ruby in Steel, 34-35****RubyMethodAttribute class properties, 492****RubyMethodAttributes class, 494-495****RubyMine, 36-37****RubyModuleAttribute properties, 483-484****Ruby on Rails. See RoR****Ruby Spec, 6****rules**

- behavior, Cucumber, 443-457
- runtime components, 24

**run method, 300**

- synchronization, 169

**running**

- assemblies, 16
- REPL console mode, 31
- RSpec tests, 437
- servers, RoR, 337
- Silverlight applications, 405
- tagged features and scenarios, 454
- unit testing, 434-435
- XAML, 307, 411-412

**runtime**

- components, 22, 23-24

**DLR. See DLR**

- ScriptRuntime class, 462-463

**S****Samples folder, 30****satisfy RSpec expression matcher, 442****scaffold command, 344, 345****Scarf library, 134****scenarios**

- Cucumber, 447-452
- hooks, 454-455
- outlines, 451-452
- tagging, 453

**SciTE, 37****scope parameter, 466****script/destroy command, 345****ScriptEngine class, 23**

- C#/VB.Net, 463-465

**Script folder, 334****script/generate command, 343-345****script/generate controller command, 346****ScriptRuntime class, 23**

- C#/VB.Net, 462-463

**ScriptRuntimeSetup class, 462****ScriptScope class, 23**

- C#/VB.Net, 465-466

**script/server command, 342****ScriptSource class, 23****sealed classes, 243****sealed methods, 250-251****searching**

- gems, 185
- living objects, 176-177
- paths, 464
- standard libraries, 159

**security**

- CAS, 19
- .NET Framework, 19

**selectors, methods, 374****self keyword, 118****semicolon (;), 44****sending messages, 283****servers, running RoR, 337****services, sockets, 149-152****Set library, 134****setter methods, 117****settings. See configuration; formatting****setup method, unit testing, 432-433****shapes, WPF, 322-323****sharing views, 380-382****Shell library, 134****shifting operators, 113****short names, assemblies, 18****Silverlight, 22, 401**

- animation, 417-418
- applications, 402-406
- chr tool, 404-406
- code, adding, 411-415
- controls, 411
- data binding, 419-422
- dynamic data, 420-421
- elements, retrieving, 412-414
- environments, preparing, 402
- event handling, 414
- folders, 30
- graphics, 415-417
- HTML, accessing, 414-415
- layouts, 410-411
- sl tool, 402-403
- static data, 419-420
- templates, 422
- web pages, adding, 406-408
- XAML, 409

**single line comments, 43****single-quoted strings, 51****singleton**

- classes, 490
- library, 154
- methods, 96
- patterns, 192-194

**Size property, 287****sl tool, 402-403****Smalltalk, 1, 5****socket library, 154****sockets, services, 149-152****source code, 29****source parameter, 407****special argument types, 225****special IronRuby methods, 231-236****special methods, 115-118****special .NET methods, 225****special parameters**

- methods, 493
- types, 93-94

**sql.rb files, 264-265****SQL Server, 260**

- connections, opening, 262
- data access, contacting, 260-265
- design, 272-276
- installing, 332

**SqlServerAccessor** class, applying, 265

**Stack** class, 38

**StackPanel** control, 317-319, 410

**StandardError**, 85

**standard libraries**, 131

applying, 131

available libraries, 132-135

MVC, 375

references, 135-158

searching, 159

**standards**

code, 47

.NET code, 211-213

REST, 339-340

**starting**, 24-25

threads, 161

**start key**, 408

**start method**, 161

**StartPosition** property, 288

**startup events**, 305

**statements**

case, 67-69

else, 81-82

if-else, 69

modifiers, 80

rescue, 78-80

writing, 44

yield, 76-77

**states, threads**, 165-167

**static classes**, 243

**static data**

binding to, 325-326

Silverlight, 419-420

**static methods**, 223, 248-249

**steps**

hooks, 454-455

implementation, 449-451

**Storyboard** element, 418

**strategy pattern**, 186-188

**String** class, 53

methods, 54, 234-235

**String.new** method, 51

**strings**

accessing, 53-54

connections

adding, 262

building, 261, 267

examples of, 261

defining, 50

delimiters, 50

double-quoted, 44-50

IronRuby code, executing from, 468-469

single-quoted, 51

time, helper classes, 349

**strong names, loading .NET assemblies**, 208-209

**structs, inheritance from CLR**, 243

**structures**

applications, WinForms, 282

classes, building, 260, 267

code file, 46-47

directories, 333-334

folders, sl tool, 403

projects, Cucumber, 445-446

Ruby

code-containing, 86

control, 64-77

**str variable**, 466

**styles**

Silverlight, 419-420

WPF, 326-327

**stylesheets, adding**, 350-351

**subscribing events**, 219-220

**substrings**, 53. *See also* strings

**succ method**, 116

**suites, test**, 433-434

**superclass method implementation, invoking**, 123

**super keyword**, 246

**suppressing layout events**, 286

**switches**

chr tool command-line, 405

command-line, 31

**symbols, Ruby**, 58

**synchronization, threads**, 167-169

**syntax**

aliasing, 215

comments, 43-44

LINQ, 279-280

Ruby, 43-47

Ruby in Steel, 35

**SyntaxError**, 80

**System.String** class, 254

**System.Windows.Forms.Application** class, 300

## T

**tables, examples**, 451-452

**tags**

Cucumber, 453-454

scenarios, hooks, 455

**target environments, .NET environments**, 482

**target machines, executing code on**, 16

**TCPServer** class, 156

**TCPSocket** class, 156, 283

**teardown method, unit testing**, 432-433

**templates**

directories, sl tool, 403

IronRubyMvs Dll files, adding, 365

Silverlight, 422

WPF, 328-329

**ternary operators**, 69

**Test** folder, 334

**testing**

Cucumber, 443-457

private methods, 428

RSpec, 435-444

**unit testing. *See* unit testing**

**Test::Unit**, 427-433

**text**

- Ruby, 50-54
- words, modifying, 357

**TextBlock element, 304****TextBlock.Loaded event, 418****Text property, 288****textual marshaling, 182****the\_next\_big\_thing method, 92****Then step, 448****third-party libraries, MVC, 375**

- references, 157

**threads, 161-169**

- exceptions within, 163
- priority, 164-165
- states, 165-167
- synchronization, 167-169

**throw\_symbol RSpec expression matcher, 442****time**

- Ruby, 62-63
- strings, helper classes, 349

**Tmp folder, 334****to\_clr\_type method, 231****ToDoListModel class, 370****tools, 30-34**

- chr, 404-406
- GUI, installing, 260
- irake, 337
- Rake, 184-185
- sl, 402-403

**-trace, 33****trees**

- expressions, 23, 24
- inheritance, 49

**types, 16**

- accessors, 107
- declaring, 48
- defining, 44
- differences, CLR and Ruby, 211
- exceptions
  - defining, 139
  - handling, 80
- special argument, 225
- special parameter, 93-94

**U****UDPSocket class, 156****unary operators, 113****undefining class methods, 491****undef methods, 96****uniform resource locators. See URLs****unit testing, 424-425**

- assertions, 428-431
- code, 426-427
- running, 434-435
- setup method, 432-433

**teardown method, 432-433****test suites, 433-434****Test::Unit, 427-433****unless, 67****unsubscribing events, 220****until loops, 70****URLs (uniform resource locators), 371****UTC difference values, modifying, 358-359****V****-v, 33****validations, ASP.NET MVC, 396-398****values**

- compatibility available, 485
- CSV, 132
- default parameter, 92
- methods, returning from, 90-91
- numeric, 48-50
- priority threads, 164-165
- SizeMode, 311-312
- return, threads, 162
- variables, configuring, 44-45

**variables**

- \$LOAD\_PATH, 210
- accessing from outside, 106-107
- classes, 103
- code standards, 47
- configuring dynamically, 178-180
- constants, 63. *See also* constants
- inside classes, 102-107
- instances, 103-104, 453
- local, 105
- PATH, 332
- Ruby, 48-64
- str, 466
- to and from IronRuby, passing, 469-470
- values, configuring, 44-45

**VB.Net. See C#/VB.Net****Vendor folder, 334****versions**

- assemblies, 18
- .NET Framework, 13, 14
- Ruby, release dates of, 6

**vertical bars (|), 96****VES (Virtual Execution System), 17****video, adding, 418****viewing**

- data templates, 423
- generated index pages, 356-357
- windows, 312

**views**

- ActionView, 341
- applications, creating, 382-385
- ASP.NET MVC validations, 397-398
- creating, 346-349

- helper methods, 380
- MVC, 339, 378-385
- results, 372-373
- sharing, 380-382

**Virtual Execution System. See VES**

**virtual methods, 245-246**

**visibility**

- control, 118-120
- initializing, 102

**Visual Studio, 35**

- designer, applying, 295-296
- extensions, creating projects, 502
- IronRubyMvs DLL files, adding, 365

## W

**-w, 33**

**web pages**

- creating, 346-354
- database-driven pages, formatting, 354-361
- resources, generating, 354-355
- Silverlight, adding, 406-408
- references, 157

**websites**

- panels, 321
- standard libraries, 159

**When step, 448**

**while loops, 70**

**whitespaces, 45**

**Window attribute, 309-310**

**windows**

- content, 315-317
- NavigationWindow, 314-315
- passing data between, 312-314
- viewing, 312
- WPF, 309-310

**Windows Forms. See WinForms**

**Windows operating system, 14**

**Windows Presentation Foundation. See WPF**

**WindowStyle attribute, 310-311**

**WinForms, 281**

- assemblies, loading, 285
- chat, building, 285-299
- Chat class, building, 282-285
- controls, adding, 289-293
- execution code, writing, 300-301
- functionality, adding, 293-295
- overview of, 281-282

**-W[level], 33**

**words, modifying, 357**

**worlds, Cucumber, 456**

**WPF (Windows Presentation Foundation), 301**

- animation, 324-325
- brushes, 322-324
- content, 315-317
- data binding, 325-329
- elements, retrieving, 308

- events, handling, 308-309
- graphics, 321-325
- IronRuby fundamentals, 307-309
- layout controls, 317-321
- overview of, 303-305
- REPL, 329-330
- shapes, 322-323
- styles, 326-327
- templates, 328-329
- windows, 309-310
- XAML, 305-307

**WrapPanel control, 318-319**

**writing**

- code, 504-506
- execution code, 300-301
- files, 158, 172-173
- Hello World!, 48
- statements, 44
- tests, 427

## X

**XAML (eXtensible Application Markup Language), 305-307**

- animation, 417-418
- elements, accessing, 412-414
- graphics, 415-417
- running, 307, 411-412
- Silverlight, 409

**-X:ColorfulConsole, 32**

**-X:CompilationThreshold, 34**

**-X:ExceptionDetail, 34**

**XML (eXtensible Markup Language) documents**

- generating, 153
- reading, 154

**-X:NoAdaptiveCompilation, 34**

**-X:PassExceptions, 34**

**-X:PrivateBinding, 34**

**XRuby, 6**

**-X:ShowClrExceptions, 34**

## Y-Z

**YAML library, 135**

- references, 157
- textual marshaling, 182

**yield keyword, 94**

**yield statement, 76-77**

**ZIP files, 27**

**Zlib library, 135, 158**