12

Extending IronRuby

This chapter covers

* Implementing Ruby idioms on C# classes
* Translating .NET naming conventions to Ruby

Conventions, as we’ve discovered, are important tools for lazy programmers; they allow us to intuit the answers to various questions and generally think less about syntax, and more on the problem we’re trying to solve. Because we’re intrepid developers wandering the borderlands of both the .NET and Ruby landscapes things can sometimes become a bit hazy on the horizon ahead.

A programmatic conversion of C# naming conventions to Ruby doesn’t always match up 100%. Every so often there will be a clash of naming conventions; C# vs Rubyisms. Good programmers will look for ways to clear the resulting pileup of prefixes, suffixes and operators. In this chapter we’ll look at how you can utilize attributes on your C# classes that provide hints on how the resulting IronRuby API should look, allowing you to provide a far more ruby-hacker-friendly interface to your classes.

Since this is a chapter about syntactic-sugar we’ll look at building a sample C# class representing a chocolate factory.

12.1 Sprinkling Some Syntactic Sugar

One of the first things you’ll be required to get to grips of when starting to develop in a new language, or even starting to work on a new project, is the code style-guide.

The style-guide provides a set of rules about how to format you source code, and importantly a set of naming conventions. Each language has a community-accepted style for naming conventions, and Ruby is no exception.

Ruby’s expressive syntax allows for some cool naming conventions (which we’ll cover shortly), and allows for method and variable names that just wouldn’t be syntactically valid in C#. This raises the problem we’ll be solving in this section: how do we write .NET classes that meet the Ruby style-guide’s naming conventions?

12.1.1 Ruby naming conventions

In order to solve the problem of disjointed naming conventions it’s good to know what exactly the Ruby style-guide is. To illustrate the differences of naming conventions let’s look at some iffy code written by our naïve Java and C# developer friend.

Variable and Method naming conversions

Ruby’s convention, as you’ve probably picked up from previous code samples, is to make verything lower case, and use an underscore to separate words.

Ruby is also flexible enough to allow you to leave off the parentheses () if your method doesn’t require any, so the convention is to leave them off to keep the code cleaner.

|  |  |  |
| --- | --- | --- |
| C# | Java | Ruby |
| MakeChocolate() | makeChocolate() | make\_chocolate |
| IsWhite() | isWhite() | white? |

In other languages methods that return Boolean values are often written with an is prefix to indicate the methods return type. Ruby’s common convention is to postfix the method with a question mark ? to indicate a Boolean return value.

12.2 Rubifying a C# class

Imagine we’d like to reuse our existing C# class, TheLandOfChocolate (home to the OompaLoompa’s that do the wonderful work of making chocolate), within IronRuby with all of Ruby’s naming conventions having been met.

In this section we’ll look at how we can provide attributes to C# that allow us to control the class’s final Ruby API, allowing us to call it as shown in Listing 12.1

Listing 12.1 Using a correctly Rubified C# class

require "Chapter12/bin/Debug/Chapter12.dll"

load\_assembly 'Chapter12.TheLandOfChocolate','Chapter12.TheLandOfChocolate'

Chapter12::TheLandOfChocolate::OompaLoompa.make\_chocolate :milk,:peppermint

12.2.1 The generated initializer

How does it work?

The attributes aren’t used at run time. They’re used by the generate initializer exec to make a file describing how to implement the API.

12.2.2 Attributes

The IronRuby runtime namespace provides a number of attribute decorators that we can use to express the class’s ruby API.

12.2.3 Ordering parameters

12.2.4 Compiling

Using