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 would be syntactically invalid in C# (like "downcase!"). 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?

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.

12.2.1 Attributes

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

|  |  |
| --- | --- |
| Attribute | Parameters |
| RubyModule |  |
| RubyClass |  |
| RubyMethod |  |
| RubyConstant |  |
| RubySingleton |  |
| RubyConstructor |  |
| RubyException |  |
| HideMethod | Hide a CLR method |
| UndefinedMethod |  |

12.2.2 Ordering parameters

12.2.3 Compiling

In this section we’ll cover the steps required to compile a CLR class into a Ruby-API friendly assembly. There are a few steps to this process.

Step 1: Add attributes

Once you’ve added attributes to your class to dictate how you want your Ruby API to be implemented you can compile the assembly. The first concept to grok is that these attributes are not directly used by IronRuby, they are only placeholders – and like a placeholder “To Do” Post-It note, they will be replaced with the real thing shortly.

Step 2: Compile your DLL

This is probably the confusing step. You’ll have to compile your DLL twice during this process. The reason you’re compiling it now is because the next step requires a compiled assembly as input to ClassInitGenerator.exe.

Step 3: ClassInitGenerator.exe

The attributes you’ve added to your code are actually used by a code generation utility, ClassInitGenerator.exe (included within the IronRuby directory).

ClassInitGenerator.exe will look for your attributes and use these to generate a new class file, Initializer.generated.cs. This file is in fact the place where the magic happens; where IronRuby gets directed how to implement the Ruby API.

Here’s an example of running the generator, creating an initializer for the Chapter12 sample code assembly.

ClassInitGenerator.exe .\bin\Debug\Chapter12.dll

/libraries:Chapter12,Chapter12.TheLandOfChocolate

/out:Initializer.generated.cs

Looking at the resulting generated code in Initializer.generated.cs you will see it’s quite complex code, and you can appreciate this attribute/code-generator approach.

Step 4: Re-compile, including Initializer.generated.cs

Pretty obviously just having a generated Initializer.generated.cs file isn’t going to have any affect unless you recompile. The generated initializer is named in a predictable format,.

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

puts Chapter12.constants.join(", ")

OUTPUTS: TheLandOfChocolate, **Chapter12LibraryInitializer**

For instance, our Chapter12 assembly will have a generated initializer class named Chapter12LibraryInitializer. This predictable class name allows IronRuby to check for the class’ existence during the load\_assembly phase.

12.2.3 load\_assembly

IronRuby’s load\_assembly method provides an easily distinguishable, shortcut way of loading .NET assemblies. It takes a full or partial assembly name as it’s parameters and attempts to load the named assembly from the GAC (or if not found there, the installation directory, or currently loaded assemblies).

load\_assembly "System.Windows.Forms"

load\_assembly "System.Windows.Forms,

Version=2.0.0.0, Culture=neutral,

PublicKeyToken=b77a5c561934e089"

Of course, in addition to loading System assemblies we can also load custom assemblies.

load\_assembly "Chapter12", "Chapter12"

TODO:More talk of second parameter. When called, load\_assembly will look for and invoke an optionally generated class initializer that can provide information that can be used to make the assembly more Ruby-like.