

MEAP

Unedited Draft

IronRuby

in Action

Ivan Porto Carrero

 MANNING





**MEAP Edition
Manning Early Access Program**

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

IronRuby in Action

- 1. Dynamic languages why should I look at them?***
- 2. Ruby bootstrapping guide***
- 3. .NET: Putting the Iron in IronRuby***
- 4. WPF: A google analytics client***
- 5. Silverlight: A twitter client***
- 6. ASP.NET: Browsing FreeBase***
- 7. ASP.NET MVC: Using Amazon for storage in the cloud***
- 8. DataAccess with IronRuby***
- 9. Extending IronRuby with C#***

1

Dynamic languages, why should I look at them?

This chapter covers:

- The differences between dynamic typing and static typing
- The differences between compiled and interpreted
- The concept of REPL
- The necessity of unit testing
- Duck typing

The first part of this book is about satisfying the prerequisites necessary to understand what's in the rest of the book. The first chapter is about some of the specific characteristics of a dynamic language and should give you a basic understanding of what it means to use a dynamic language. We'll cover some of the essential building blocks for using the Ruby language throughout this book.

A couple of years ago I wanted to learn a new language because I felt a bit constrained in the one that I was using at the time. Some of the blogs I was reading were from people that programmed in completely different languages than C#. These people did some really cool stuff, and I wanted a piece of that. The most natural step for me at that point was to get heavily into JavaScript. Not long after that there was an AJAX revolution that hit the web developer community. JavaScript was dynamic and suddenly I was writing pretty complex things with a lot less code than I would have to do in C#. In short I absolutely loved it. At the same time a friend of mine had just discovered Ruby and was full of love over it, he spoke of beautiful and concise code as well as "things just work". I decided I wanted to see that for myself, and sure enough a couple of days later I was absolutely hooked because its syntax was very "human-friendly" as well as the fact that I could manipulate the program at runtime. With a little Ruby exposure my C# started to look completely different and ultimately I became a much better programmer in any of the languages I knew then. I hope I can transfer some of my enthusiasm for this pretty language to you through this book.

In our experience many programmers confuse the different typing systems and because of this reach wrong conclusions on how types are treated. And that's why we felt it was important to start with a discussion on that topic.

Another topic of discussion will be REPL. REPL stands for Read-Evaluate-Print-Loop, which is a process that consists of entering some code in console, evaluating that code, and printing the results of that evaluation, at

which point the execution in the console loops back to taking input. REPL is a beautiful way of programming a.o. because of its really short feedback loop and flexibility; this should give you a primer on what REPL is.

Many languages provide mechanisms for runtime code generation; we'll contrast the complex API in our static language with a much simpler mechanism of string evaluation.

With all of that dynamism, we would like some reassurance that our code works, and that brings us to unit testing. We'll see how that can be done using IronRuby with the tools from the Ruby community. But let's look at some of the vocabulary we'll be using in this book.

1.1 Setting some ground rules

We would like to provide some clarification for some of the vocabulary that will be used in this chapter as well as to give you our view on why you should learn new languages on a regular basis. For the purpose of this book we will plead in favor of learning a dynamic language but we do believe that all language types are beneficial to your evolution as a programmer.

For example, a novice programmer should start by learning a statically typed and compiled language (like C++, C#, Java, Visual basic) because those languages force you to take a lot of care and are less forgiving about mistakes. That should give you a much better basis to learn other types of languages from.

1.1.1 Some terminology explained

In this chapter we will use terms like CLR, DLR, Ruby, IronRuby, TDD and BDD. In this section we'll briefly explain what these terms mean and tell you where you can find the more detailed description in this book or on the web.

We're assuming that you know what the .NET framework means. The CLR is a part of the .NET framework and indicates the Common Language Runtime which is the virtual machine in which a bytecode form of the Common Intermediate Language (CIL/IL) is run. The CIL is the lowest-level human-readable programming language in the .NET Framework. We delve deeper into the CLR in chapter 3 where we outline the technologies we'll be using in this book.

The DLR (Dynamic Language Runtime) is a runtime environment built in C# that runs on top of the CLR and enables a lot of mechanisms for dynamic language developers to implement their language on the .NET framework. Some of the languages that have been implemented on the DLR include (Iron)Python, (Iron)Ruby, (Iron)Scheme,... In the case of IronRuby the Iron simply stands for the fact that it is the Ruby language implemented on the .NET framework. We will use IronRuby and Ruby interchangeably throughout this book but we will always be talking about IronRuby in the context of this book. We'll also look closer at the DLR in chapter 3.

The last terms we need to explain are TDD (Test driven development) and BDD (behavior driven development). Test driven development is a method of developing software where you first write a test for the piece of functionality you're going to implement and then write the code to satisfy that test. At the end of your development you will have a battery of tests that aid you in maintaining your application and give you a huge confident boost about your code. For more information on test driven development we'd like to give <http://testdriven.com> as a starting point.

Behavior driven development builds on the knowledge of test driven development but extends it by questioning what the behavior of an application should be both before and during the development process. Behavior driven development expresses those in a more textual form than its predecessor. A good starting point on behavior driven development is <http://behaviour-driven.org/>.

In the next part of this section we'd like share our views on why learning a different language is a good thing. This paragraph will give you some of our personal reasons for getting into Ruby some time ago.

1.1.2 Why learn a different language?

The reason for putting this part in the first chapter is because programming language debates can get quite religious and vile at times. I would like to encourage you not to partake in any of these debates because I think they are futile and serve no purpose but to antagonize. By this I don't mean you should shun all discussion around them, I just want you to use some judgment and not get into a "my language is better" debate... I would like to make it clear to you that every language has its benefits and disadvantages; I would also like to convince you that you will become a better programmer with every language you pick up even if you never get to use them in a production scenario (typical examples of these languages are LISP, Haskell,...).

In the next part of this chapter I will try to clarify the confusion surrounding the different typing systems that are being used when talking about programming languages. But for now let's get on to why I started learning a dynamic language. In this paragraph I will be making a plea for learning a dynamic language.

I had been programming C# for a couple of years and before that I had been using turbo pascal, both are statically typed and compiled languages in covariance and contravariance scenarios. In C# I loved the introduction of generics but they weren't always able to give me the flexibility that I needed. I wanted more and at the same time there was murmur about Ruby on Rails; being a web developer I got curious and started an investigation into the Ruby language and the Rails framework. My friend, Alex, showed me the IronPython project and I borrowed a book on python for a couple of days. I still couldn't make my mind up whether or not I should go for Python or for Ruby. So I went to a bookstore and bought a couple of books on both languages. Amongst those books were the Python cookbook and the Ruby cookbook.

At this point I knew that with both languages you could do pretty much the same, but the syntax of Ruby appealed more to me. And because I had gone through the two cookbooks I also came to the realization that those languages were incredibly powerful tools. I decided to dig deeper and started playing around with Ruby more and more. Ruby felt very much like JavaScript which is another one of my favorite languages. I have objects and everything is essentially behaves as a hash. I could add methods and properties to those objects at runtime. In short at that moment I was past the point of no return. I had to find out all I possibly could on becoming a good Ruby developer.

The more I got into Ruby the more I wanted to gain some of the same productivity boosts and "genericity" in my C# programming. I found new and faster ways of doing things which made me ultimately a better programmer in the C# language as well. The fact that Ruby had a console where I could quickly try things out by just typing them was too easy and too much fun to just leave it at that.

The fact that a dynamic language often results in hugely reduced code volume. That alone has a couple of very important benefits. The first benefit is less typing, which means greater development speed. A second reason is that with less lines of code there is less chance of having bugs. A reduced code volume is easier and faster to code, debug and maintain that is basically what I'm getting at.

In my professional experience I would like to learn a new programming language every year. Because every time I do so I learn so many new tricks. And everybody knows that a one trick pony isn't the best value for your money; the more you know, the better your programs will be.

If you want things to be absolutely fast and then you would probably use C++ for that application. When a really good C++ programmer writes his code it will run faster than managed code (code that runs on the CLR). The speed boost you get is generally only about 5-10%. However if speed doesn't matter that much you can use a dynamic language, for example, which will enable you to complete that application in half the time with a lot less code than say in C#, but there is always a trade-off you make. It just depends on the situation which trade-off is the most valuable to make.

And this concludes the ground rules section of the book. This section should give you a good basis for continuing the rest of this chapter and book. In the next section we'd like to hand you some of the essential building blocks you need to understand what it means for a language to be a dynamic language.

1.2 Essential building blocks

If you're already familiar with Ruby this chapter should give you an understanding on how most of the classic languages in the .NET framework have been implemented. It should also make you a little familiar with what you would have to deal with if you decide to extend the IronRuby implementation. If you're already familiar with C#; our example of a statically, strong typed and compiled reference language; this chapter should give you an understanding of the basic differences and usages behind the typing systems.

But before we can get to some coding, we think a more high level discussion of some of the basic differences between dynamic and static typing is in order. This will give you a better understanding of what goes on in the background. Next, it's our belief that we should take a look at the differences between a compiled and an interpreted language. This will deepen the understanding and fortify your foundation for grasping some of the apparent magic that seems to be going on behind the scenes.

Don't worry too much about understanding some of the code listings that we'll be using to demonstrate our point. Most of it will be explained later in much more depth. Let's get started, shall we?

1.2.1 Compiled and interpreted languages

First we'd like to explain what a compiled language means and what an interpreted language means. Technically it should say languages implemented either through a compiler or through an interpreter.

A compiled language in the context of this book is a language that requires a compiler before it can be executed. VB.NET, C# and Java all require a compilation step before they can be executed. This requires that the data type of every variable, function parameter and function return value has to be known at compile time through declarations. Because of this there is the benefit that the compiler can check for any syntactical errors that may have snuck into your code. It also checks for invalid constructs and values that may have crept into your code. They don't however give you the guarantee that your program will work as you intended it to work. Because of the compilation step, the compiler may have made some optimizations. This implementation is generally faster at runtime. Because of the fact that the compiler checks all the types in your application sometimes people also call it a type-safe language. This is only partially true because interpreted languages can also be type-safe which is exactly what this chapter tries to make clear.

An interpreted language means that the code you feed it; gets executed by an interpreter, whose job it is to analyze each statement it encounters and consequently execute the desired action. There is no error checking at compile time because there is no compile time. This means that your development experience in general is more pleasant because you can have a code - run - debug cycle instead of a code - compile - run - debug cycle. Another advantage of an interpreted language is that the interpreter has a lot more information available when it analyzes the code because it knows exactly what its run-time environment is like and the interpreter also has access to everything that is loaded into memory and as such the interpreter can make better and more complex decisions.

Although the classic .NET languages like VB.NET, C#, C++.NET all are languages that are compiled, they are only compiled into bytecode which is not native code. This bytecode gets transformed to native code through a process of Just-In-Time compilation when the code gets first executed at run-time. So the compiler has as much information about its run-time environment as an interpreter at a small cost the first time some code is executed, this approach preserves the advantages of languages implemented through a compiler.

Interpreted languages still have a couple of distinct advantages even over the classic .NET languages. Because of their nature, programs written in a dynamic language can modify themselves at run-time like adding methods and properties, changing methods, generate new classes etc. They generally have a first class eval function that can evaluate strings into executable code. This interpreted nature gives the possibility to test the code you write into a console and execute it.

Suppose you have to talk to a business partner or so who speaks a different language than you. You need to give him a list of tasks (the program) but you need a third party to translate that information.

In the interpreted scenario you would take a translator to the meeting and he would translate a question on the spot to your business partner. Your business partner would then answer that question and wait for you to give him a next question.

In the compiled scenario you would contact a translator beforehand and have him translate all the questions. You would then meet with your business partner and hand him over the list of questions. He will then go off and answer all the questions on that list before sending them back to you.

Next we'll talk about the typing systems used in programming languages, which is another essential building block for a programming language. Understanding that concept properly allows you to have a clearer picture of the constraints of your language.

1.2.2 Typing

We're touching on the subject of typing here because it's often a source of confusion. The next couple of paragraphs aim to clarify some of the common confusions around typing. We'll talk about static, dynamic typing; type inference; strong and weak typing; and, lastly, duck typing.

STATIC, DYNAMIC TYPING AND TYPE INFERENCE

The process of declaring the variables, function input parameters and function return values is called static typing. A statically typed language does not necessarily require you to declare the variable close to its use, but you do

have to declare them before you use them. Types can be cast or converted to another type. The classic .NET languages are all statically typed, where Visual Basic.NET allows you to optionally leave out the declaration for function parameters, and the compiler enforces this. The advantage is that this code can execute more quickly. The disadvantage of this approach is that you have a lot more keystrokes to do when writing a program.

We'll illustrate this point in listing 1.1. It shows you some of the most basic things you can do with a statically typed language.

Listing 1.1: Static typing in c#

```
int i = 5;
i += 5;

System.Console.WriteLine(i); //→ 10                                1
// i = "hello";                                                    2

string coerced = "" + i; //→ 10
string converted = i.ToString(); //→ 10

System.Console.WriteLine("Coerced: {0}", coerced); //→ Coerced: 10
System.Console.WriteLine("Converted: {0}", converted); //→ Converted: 10
```

- 1 Works because of appropriate overload**
- 2 Won't compile, i is int not string**

In the example above (listing 1.1) we first declare an `int i` with a value of 5 and next we add 5 to that int. After which we write the result to the console window, we can do that without a conversion because the `Console.WriteLine` has an overload that takes an `int` [#1]. The following line is commented out because it wouldn't compile otherwise but it illustrates that you can't change the type of `i` to be a string [#2]. Next we take advantage of some of the type coercion that `c#` understands, we also convert the `int` value explicitly to a string and we output the results to the console window.

The languages built on top of the DLR (we'll talk more about this in the next chapter), are dynamically typed languages. One way to make a distinction is to say: a language is dynamic when it doesn't require variable declarations before they are used. You could also make a distinction between the two typing methods by at which time the type checking occurs; this is more accurate in our opinion because both typing mechanisms have casting and conversion techniques. This distinction would then be that statically typed languages perform most their type checking at compile time where dynamic typed languages defer all of that checking to run time.

Dynamic typing may allow compilers and interpreters to run more quickly, as well as save a lot of time during development because you don't have to deal with the tedious work of declaring every single step along the way.

Modern static languages like `C# 3.0` understand the concept of type inference. Type inference occurs when you don't have to specifically tell the compiler what type your variable will be but it will instead be inferred on first assignment. From then on that variable is of the type assigned to it and that type can't be changed anymore. Type inference is different from dynamic typing in the fact that a dynamically typed variable can still change its type during the execution of the program. An example of type inference in `C# 3.0` is illustrated in code listing 1.2.

Listing 1.2: Type inference in C# 3.0

```
var i = 5;

i = i + 5;
System.Console.WriteLine("Inferred type: {0}, value: {1}", i.GetType().Name, i);           A

// i = "hello";                                                    B

System.Console.WriteLine("Converted type: {0}, value: \"{1}\"", i.ToString().GetType().Name, i);
```

A Valid operation after assignment
B Won't compile, i is int not string

Type inference still checks at compile time whereas dynamic typing defers its type checking to runtime. Listing 1.3 illustrates the same basic operations; it just adds one operation because the type of a variable can be changed at runtime.

Listing 1.3: Dynamically typed in IronRuby

```
a = 5
a = a + 5

#the following is valid because a number
#implements a to_s method that will be called
#to convert this value to a string
puts "the type: #{a.class.to_s}, the value: #{a}"

a = "123"

# This is valid because a is a string and has support for the length method
puts "the type: #{a.class.to_s}, the value: #{a.length}" # outputs String, 3

# The following is invalid because a number doesn't know how to perform an
# addition with a value of type string so it will throw a TypeError.
# And the program stops working at this point because of the invalid type
a = 5 + a
```

Static and dynamic typing is different from strong and weak typing. People use these terms interchangeably but they are very different concepts. It's possible for a language to be both dynamic and strong typed. The same goes for a static typed language that is also weak typed.

STRONG AND WEAK TYPING

A strongly typed language is a language where every variable has a specific data type. As such all the operations that are allowed against that variable are known upfront. So a strong typed language would be a language that only allows safe operations against its variables; where a weak typed language implicitly casts variables to other types.

The deciding factor here is whether the language implicitly converts unrelated types without warning, allowing you to add the integer 1 to the string "10" and arrive at the result "110" or 11, depending on how you write your statement. Strongly typed languages will cry foul where weakly typed will languages simply do what they think you mean and continue.

Strong typed are generally easier to debug than their weak type variants. An example of a weak typed language is C or Perl. Both C# and (Iron)Ruby are strong typed.

DUCK TYPING

An explanation of duck typing wouldn't be complete without the obligatory quote: "If it looks like a duck and quacks like a duck, it may as well be a duck." This quote just begs for some explanation: Duck typing means that a parameter on a method only has to respond to the methods and properties that are being used by the method with the parameters. In other words if a method has a parameter `variable` and inside its body it uses the method `some_method` on that parameter `variable` then any type that implements `some_method` is a valid type.

Duck typing considers the methods to which a value responds and the attributes it possesses of bigger importance than its relationship to a type hierarchy. This encourages greater polymorphism because types are enforced as late as possible.

In a nutshell, if your method takes a parameter and your method calls the method `print` on that parameter than the users of that method can pass any type to that method as long as it has a method `print` it'll be ok for the callee.

At the point of this writing IronRuby is a strong and dynamically typed language that supports duck typing and is implemented through an interpreter on the DLR and CLR. That makes it slower than its compiled brothers and sisters in the .NET language pool, but not that slow that you shouldn't use it.

1.2.3 Why Ruby's type system is a good thing

Ruby makes more than up for its slower speed by representing a kind of best of breed implementation of a programming language. Ruby borrows the best features from Smalltalk to Java and Perl to Python. Because of its nature it allows you to develop applications quickly and with very expressive code.

The Ruby language has more attributes than just the ones mentioned above. You can use features from functional programming, it understands closures, it allows for objects to be altered at run-time and it supports introspection.

There is an ongoing discussion about whether static typing does actually give you more safe code, after all type errors are the most trivial of errors and usually pretty easy to fix. Even in statically typed languages we often use untyped code, like passing Object around instead of a more concrete type. And yet that doesn't yield as many errors as expected because you typically know which behavior to expect in a given situation.

Next we'll look at some of the consequences of having an interpreted, dynamic language at your disposal. We'll cover the console with REPL and talk about the first class eval method in comparison to the Reflection.Emit API that can be used in C# to execute late bound code. After that we'll take a look at why unit tests are of huge importance when using a language like IronRuby. We'll also briefly touch the tools that you can use with IronRuby to use unit testing.

1.3 Consequences of an interpreted, dynamic language

By now you must be in dire need of some action. First, we need to make sure that we have all the prerequisites necessary for continuing the rest of this book. At the time of this writing you couldn't download any binary packages of the IronRuby implementation. So we'll talk you through the steps of getting the source and compiling it.

After that we'll get to do a little bit of coding using the interactive console, at which time we'll introduce you to the concept of REPL and why the console will be your new best programming buddy. When you're familiar with the console it's time to look at what it means to have a first class eval method in your programming language.

1.3.1 Bootstrapping your environment

Let's get down to business. In this chapter we'll explain how you check out the source code of IronRuby. After that we'll compile the IronRuby interpreter and we're set to start developing. To be able to compile the source code we chose for the option to let rake build it. So we'll also be getting the C-Ruby implementation.

You can get the C-Ruby version from <http://www.ruby-lang.org/en/downloads/>. Download the one click installer and when the install is completed add C:\Ruby\bin to your PATH environment variable. Next type `gem install rake pathname2` in a newly opened console window, as demonstrated in figure 1.1

```
+ C:\Users\Ivan Porto Carrero
>> gem install rake pathname2
Updating metadata for 34 gems from http://gems.rubyforge.org
.....
complete
Successfully installed rake-0.8.1
Successfully installed pathname2-1.5.2
2 gems installed
Installing ri documentation for rake-0.8.1...
Installing ri documentation for pathname2-1.5.2...
Installing RDoc documentation for rake-0.8.1...
Installing RDoc documentation for pathname2-1.5.2...
```

Figure 1.1: The PowerShell window with the gem command

In addition to rake you, will also need a C# compiler. This comes with the visual studio installation or with the Microsoft .net framework SDK. Now, on to IronRuby...

The source code for IronRuby can be downloaded from rubyforge <http://IronRuby.rubyforge.org>. To download the source code you need a subversion client. My client of choice is TortoiseSVN on Windows. On Macs or Linux you can use a plug-in for Eclipse to accomplish this. The process of downloading is checking out the code. That will

enable you to get updates as they are being pushed in the repository. The url to check out is <http://IronRuby.rubyforge.org/svn/>. To check IronRuby out from the repository, open an explorer window and navigate to where you want to save the source code. Next, right-click in an empty area of that folder and choose checkout from the context menu. That will bring up the tortoise checkout window (figure 1.2).

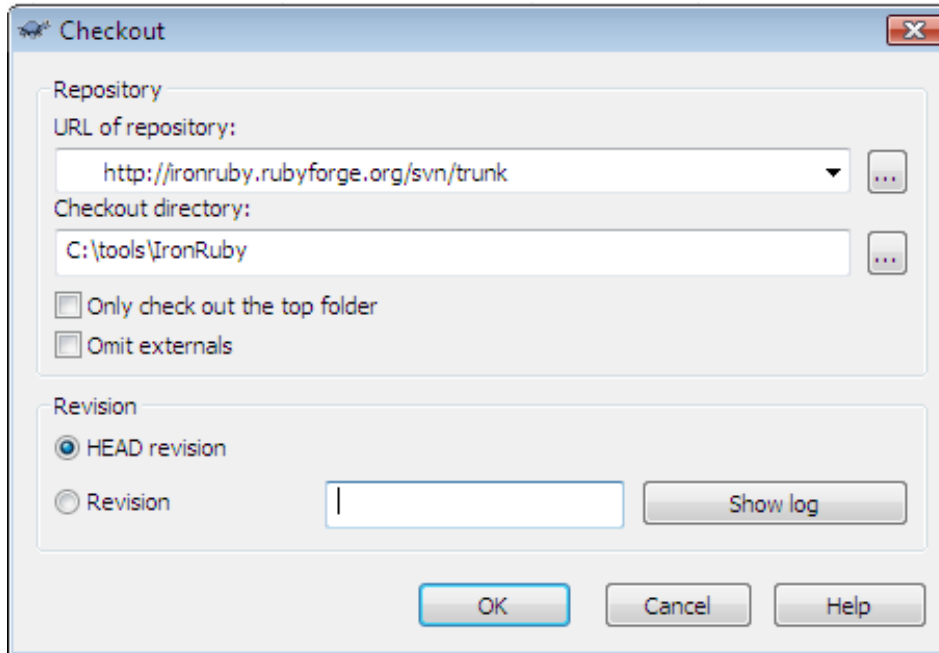


Figure 1.2: Checking out the IronRuby source code with tortoise

You now have all the tools to compile IronRuby. To do this, simply open a console window and navigate to the location of the IronRuby source code. We're going to use rake to build the IronRuby language and interpreter. Type the rake command at the command line and that should compile IronRuby. You can specify amongst other things which configuration you want to build: debug or release. I'll show you how to compile the debug version as displayed in figure 1.3.

```
+ C:\
>> cd \tools\ironruby
+ C:\tools\IronRuby
>> rake compile config=debug
<in C:/tools/IronRuby>
Read in 17 resources from "C:\tools\IronRuby\src\microsoft.scripting\Math\MathResources.resx"
Writing resource file... Done.
Read in 48 resources from "C:\tools\IronRuby\src\microsoft.scripting\Resources.resx"
Writing resource file... Done.
```

Figure 1.3: Compiling the debug version of IronRuby with rake

TIP

Add the path to the rbx executable to your PATH environment variable.

Woohoo! Step 1 of getting started with IronRuby is complete. You now have IronRuby. Let's find out what we can do with these newly gained abilities. We'll first look at the interactive console and how it enables us to take advantage of REPL.

1.3.2 REPL: The console, your new best friend

REPL is a really convenient method of developing software. With a compiled language your code is effectively dead until you compile it and start up the binary. At that moment your code is working for you but you have little or no control over it. Sure you can get into the debugger and use a nifty feature like edit and continue but the possibilities that that gives you are fairly limited.

Many dynamic languages have a concept of REPL. We'll first explain what REPL means and how it can be of use to you after which we'll look at a couple examples of using REPL in practice.

REPL

REPL often manifests itself as a type of interactive console in which you type a line of code. As you may recall from our earlier discussion, that line of code will be **read** by the console; then **evaluated** by the interpreter and the yielded result will be **printed** in the console after which the console will **loop** to read the next statement you feed it.

The moment I understood the beauty of this development method, programming Ruby became a lot more interactive and interesting for me. I could see immediately that having such a console was a fantastic way of learning a new language. This is quite useful for prototyping and experimentation, don't you agree? IronRuby does REPL, why don't we try it out?

THE INTERACTIVE CONSOLE, A GOOD PLAYGROUND

The interactive console lets you interact directly with the Ruby interpreter to try out and test little bits of code.

When you're using REPL, you don't write your code first and load it later. Instead, you enter your code piecemeal, function by function, variable by variable at that innocent looking prompt. You develop incrementally, and at every single moment, your objects and functions are alive. You can access them, inspect them and even modify them. Your code becomes this living thing you are interacting with. It almost doesn't feel like programming, it feels more like experimenting or playing.

This is the first time that we'll look at a "Hello world" implementation. Actually it's only a one line implementation and we're going to use the interactive console to test it. So let's spin that console and get busy.

To launch the console open a command window or powershell prompt and navigate to the build/debug folder in the folder where you downloaded the IronRuby source code. Type `./rbx` to start up the console (see listing 1.4).

Listing 1.4: Starting the console

```
PS C:\IronRuby\build\debug> ./rbx
IronRuby Pre-Alpha (1.0.0.0) on .NET 2.0.50727.1433
Copyright (c) Microsoft Corporation. All rights reserved.
>>>
```

Now at the prompt, type "Hello world" and hit enter (listing 1.5). Oh, that seems to have done something. Well that would be correct but nothing has been executed. #1 The sign `=>` is a way for the console to tell you the result of the last expression that has been validated.

Listing 1.5: Entering a string value in the console

```
PS C:\IronRuby\build\debug> ./rbx
IronRuby Pre-Alpha (1.0.0.0) on .NET 2.0.50727.1433
Copyright (c) Microsoft Corporation. All rights reserved.
>>> "Hello world"
=> "Hello world"
>>>
```

A

A console indicating a string was created

But for the console to really do some work it needs a little bit more than just a string it needs an instruction as well. In order to feed it an instruction, the instruction of choice would be `puts` (similar to `System.Console.WriteLine` from C#). Listing 1.6 should be the complete implementation of Hello world in the console.

Listing 1.6: Finishing the implementation of Hello world in the console

```
PS C:\IronRuby\build\debug> ./rbx
IronRuby Pre-Alpha (1.0.0.0) on .NET 2.0.50727.1433
```

```
Copyright (c) Microsoft Corporation. All rights reserved.
>>> puts "Hello, World!!!"
Hello, World!!!
```

A

A actual output (no =>)

And that's all it takes to write hello world inside the console. Later on we'll do some more work in the console but first we would like the chance of pondering a bit on what it means exactly to have a first class eval function in a language and how it will ease the pain of doing dynamic development. There are a couple other ways to run ruby programs with the .NET Framework, which is what we'll look at next.

RBX CAN BE USED IN MORE WAYS

All that interactivity is all fine and dandy but how does that make your code repeatable? Let alone running it on different computers?

Of course the console is not the only way to execute Ruby code. You could place it in a file with extension .rb or any other textfile for that matter. Calling the command `rbx <<filename>>` will execute all the code contained in that file.

Let's just do that with our helloworld implementation from listing 1.7. If you still have the console open you can copy the line that says `puts "Hello, World!!!"` and paste that into a new text file. If you save that file with name `helloworld.rb` in the same folder as where the `rbx.exe` file lives we should be able to execute that file by executing the following command: `rbx helloworld.rb`. That should return the output shown in listing 1.7.

Listing 1.7: The output of running the helloworld program.

```
PS C:\IronRuby\build\debug> rbx helloworld.rb
Hello, World!!!
```

You can also host IronRuby inside other .NET applications to execute ruby code. This topic will be discussed in more detail in chapter 3.

Those are the different ways you can run Ruby programs. So far we've discussed the type of language that Ruby is and we've also touched on how to run Ruby programs. The console comes with a whole set of command-line parameter which is what we'll be looking at next.

THE CONSOLE AND ITS COMMAND-LINE PARAMETERS

Like any self-respecting console application; the IronRuby console application `rbx.exe` has a bunch of command-line parameters to customize the behavior of the console. We'll have a closer look at those parameters because they can be quite helpful when developing or optimizing your code.

There are 2 types of categories in the parameters. The first category of parameters deals with the actual scripting host and the second category deals with the execution of the scripting engine. We decided to talk about these parameters in this chapter although some concepts will still be very foreign to you most of them will be clarified in chapter 3.

We previously mentioned that IronRuby is based on the DLR, which is a base for implementing dynamic languages on the .NET Framework. This implies that there is a more general console for working with the DLR and that is what we called the scripting host, which is a console application that will host the scripting engine in this case. Table 1.1 shows an overview of the command parameters and offers a short explanation of what the parameter does. In the case of the `rbx` command some switches won't work because this application is specialized for IronRuby.

Table 1.1: Overview of the valid command-line parameters for the scripting host

Parameter	Description
<code>/help</code>	Shows an overview of all the parameters with a short description, including the non-working switches
<code>/run:<files></code>	Runs the specified semicolon-separated list of files one by one via the IronRuby scripting

	engine
/execute:<file>	Executes an exe file using it's static entry point
/paths:<file-path-list>	To be used with /run only and specifies a semicolon separated list of import paths
/mta	Starts the scripting engine in a multi-threaded apartment state. This isn't available in Silverlight
/setenv:<var1=value1;...>	Sets the specified environment variables for the console process. This isn't available in Silverlight
/X:ShowASTs	Shows the generate Abstract Syntax Trees in the console. We will return to ASTs in chapter 3
/X:DumpASTs	Saves the Abstract Syntax Trees as files in the current directory. We will return to ASTs in chapter 3
/X:ShowRules	Shows the generated action dispatch rules in the console.

Table 1.2 shows the parameters that relate to the `RubyEngine`, which we referred to earlier as the scripting engine. The scripting engine holds the implementation of the IronRuby interpreter in our case.

Table 1.2: Overview of the command-line parameters for the scripting engine

Parameter	Description
-c <command>	The command is a ruby string that will be executed as a program. This option terminates the option list
-h	Displays the usage of the scripting engine. This are these command line switches
-i	If you pass a ruby file to the command with this switch it will end in the console mode so you can inspect variables etc.
-V	Print the version number and exit
-D	Enable application debugging
-X:AutoIndent	if you're writing classes or multi-line blocks in the console this will indent those automatically for you.
-X:AssembliesDir	Sets the location for saving the generated assemblies. To be used with <code>-X:SaveAssemblies</code>
-X:ColorfulConsole	Enable colors in the console
-X:DumpASTs	Shows the generate Abstract Syntax Trees in the console. We will return to ASTs in chapter 3
-X:ExceptionDetail	Enables full exception messages, these may generate really long errors and may or may not be more useful than the standard errors
-X:Interpret	Enable interpreted mode. This means that most optimizations have been turned off.
-X:Frames	Generate custom frames. When custom frames are turned on you have access to all the local variables in an environment through a dictionary. This comes with a performance penalty for not using the real storage mechanism for locals.
-	Use tuples as the internal storage mechanism for optimized scopes
X:TupleBasedOptimizedScopes	
-X:ILDebug	Save the generated IL output to a text file for debugging. We'll talk a bit more about this in the next section.
-X:MaxRecursion	Sets the maximum recursion level that you want to allow in your code
-X:NoTraceback	Don't emit traceback code. When running in debug mode you normally get a dynamic stack

	trace, this option disables that stacktrace
-X:PassExceptions	Do not catch exceptions that are unhandled by the script code
-X:PrivateBinding	Enable binding to private members
-X:SaveAssemblies	Saves the generated assemblies, to be used in combination with -X:AssembliesDir
-X:ShowASTs	Shows the generate Abstract Syntax Trees in the console. We will return to ASTs in chapter 3
-X:ShowClrExceptions	Turns of the IronRuby specific error messages and uses the CLS exception information instead
-X:ShowRules	Shows the AST for the generated action dispatch rules in the console.
-X:SlowOps	Enable fast ops
-X:TabCompletion	Enable tabcompletion mode in the console
-X:TrackPerformance	Track performance sensitive areas (only available in debug mode)
-X:CachePointersInApartment	Cache COM pointers per apartment (only available in debug mode)

The next discussion we would like to have in this essential building blocks section is having the language feature of functions as first class citizens and more importantly having a good eval function.

1.3.3 Eval: trading in Reflection.Emit for simplicity

If you're an experienced .NET developer, you've probably used reflection on a couple occasions. And sometimes you may have wished that there was a way for you to generate some code and then have that code participate in your program. In the static languages of the .NET framework you are kind of able to, although you would have to know about the shape your type upfront for it to fully participate in your code.

It's a little bit unfortunate but we're going to have to deep dive into some of the corners of the .NET framework. Don't be alarmed or intimidated by the fact that you have not seen these abilities before or you haven't used .NET before. We just need to do this to contrast the differences in the 2 approaches. On the other hand it might be a nice way for you to discover a lesser known feature of the C# language.

We'll look at an implementation that uses the .NET facilities for doing runtime code generation and subsequently we'll look at an implementation in the Ruby language. We'll turn to the good old hello world program so that it is simple enough for everyone to understand.

REFLECTION.EMIT: EXECUTING ARBITRARY CODE THE HARD WAY

When you compile your code in VB.NET or C# or in any other compiled language that is implemented on top of the .NET framework; the code doesn't compile into native code instead it compiles into CIL (Common Intermediate Language) bytecode. And right before a method gets executed for the first time at runtime it gets compiled into native code. This is a good way of doing something because there is only a slight delay when a method gets executed for the first time. But the native code that is generated can be heavily optimized by the compiler because it has a lot of information about the platform it is running on. This also means that you could generate IL to do some code generation at runtime. Let's look at what happens when we implement hello world.

To write our hello world implementation in C# we would do have to write something like listing 1.8

Listing 1.8: Hello world in C#

```
public static void Main(string[] args)
{
    System.Console.WriteLine("Hello, World!!!"); //outputs Hello, World!!!
}
```

And when we compiled that bit of code it could be a bytecode representation like listing 1.9

Listing 1.9: Hello world in IL

```
.method public static void MyMain() cil managed
{
    .entrypoint
```

```

    ldstr  "Hello, World!!!"
    call  void [mscorlib]System.Console::WriteLine(string)
    ret   //outputs Hello, World!!!
}

```

Now when we would want to generate that bit of code at runtime instead of having it precompiled we can use the C# code from listing 1.10.

Listing 1.10: Generating code at runtime with C#

```

public class HelloWorldLCG
{
    public static void Demonstrate()
    {
        DynamicMethod dm = new DynamicMethod("HelloWorld", typeof(void), new Type[] { },
        typeof(HelloWorldLCG), false);

        ILGenerator il = dm.GetILGenerator();
        il.Emit(OpCodes.Ldstr, "hello, world");
        il.Emit(OpCodes.Call, typeof(Console).GetMethod("WriteLine", new Type[] {
        typeof(string) }));
        il.Emit(OpCodes.Ret);

        dm.Invoke(null, null);
    }
}

```

The point we're trying to make here is that doing dynamic code generation in C# isn't the easiest thing, but it can be done. You would have to understand IL pretty well to make extensive use of that feature of the .NET framework. We didn't talk about leveraging the CSharpCodeProvider way of doing this because the comparison isn't really valid. We're talking about generating code and executing that in on the fly without having to generate and compile a file first. When using the CSharpCodeProvider you also have to load the compiled assembly etc. Next we'll look at how to achieve this run-time execution of code in IronRuby.

EVAL: EXECUTING ARBITRARY CODE MADE A LOT EASIER

Many programming languages have the notion of an eval function. An eval function is a function that takes functions expressed as Abstract Syntax Trees or text to the execution engine or runtime and have those executed. IronRuby has such an eval function, and for now we only need to know that it works. We'll return to this function in more detail later. But it's important for you to know that it exists because we may be using it in some places later in the book.

Since IronRuby itself is implemented on top of the DLR (Dynamic Language Runtime) and the DLR is implemented on the CLR (Common Language Runtime) which is the runtime environment of C# it shouldn't surprise you that our `puts "hello, world"` implementation ultimately also generated IL that was executed by the CLR. And that that generated IL is pretty much the same as the one you needed for the C# hello world implementation.

Now suppose we would want to generate that method at runtime, like in our C# example, well in that case we would have to type the following code: `eval('puts "hello world"')` (listing 1.11) in our console to see that result. Listing 1.11 is currently included as MRI because eval doesn't work yet in IronRuby.

Listing 1.11: The Ruby way of dynamically generating code

```

PS C:\IronRuby> irb
irb(main):001:0> eval('puts "hello world"')
hello world

```

I don't know about you but that seems a lot easier to me than using the Reflection.Emit API. There are some risks and valid uses of the eval function but more about that later.

This section dealt with some of the tools and characteristics of a dynamic language, so far it has all been good news I think. But using a dynamic language also has some consequences especially if you're coming from languages like C#, VB.NET, Java, C++. We'll look at some of the consequences of working with a dynamic language, like making sure your code behaves as expected and why you don't need Interfaces in Ruby.

1.4 The reality of using a dynamic language

We believe every software developer tries to achieve building maintainable, reusable and flexible code. Some of the concepts that spring to mind are loose coupling, component-based programming, contract first design etc.

A good practice in development is to write unit tests to prove that your code behaves as you intended it. We'll talk briefly on why that is even more important for a dynamically typed language than it is for a statically typed language. We'll look at which unit testing tools are available for IronRuby and how to use them. Unit tests will be used everywhere throughout the book so we thought we'd talk about them right at the beginning.

In C# for example you can only inherit from one base class but there is a concept of interfaces which allows you to work around the lack of multiple inheritances. A term often used is interface-based design where common behaviors are grouped into interfaces. Our discussion will try to convince you that in duck typed languages there is no need for interfaces.

1.4.1 Unit testing

Unit testing is a tool that helps you in the first place write maintainable software. Over the last couple of years it has become a common practice to write tests first for a method you're going to write and then write the code in that method that satisfies those tests. Unit testing doesn't guarantee you that your code is correct but it can give you enough confidence, an approximate proof within reasonable error bounds, in its correctness. The more tests the smaller the error bounds and the better your confidence.

Writing unit tests helps you in several ways to write better code, because it forces you to think about the code before you write it. It helps you to write more loosely coupled code because that is something that flows naturally when writing tests first. It helps you to verify that the code you write is accurate by providing an instant feedback mechanism during the development process. It also helps you after the code is written because you can verify if any changes you made broke something in your program by running the unit tests. And it can also serve as a form of documentation for your code because you're using the code in your tests.

UNIT TESTING IS A GOOD THING, EMBRACE IT.

To do this properly requires some discipline on your part, the code still won't write itself. In Ruby unit testing is very important because of the dynamic nature of the code. You're very able to make small changes that have huge consequences for your codebase. Because of this you will want to have a good test suite to ensure that you can refactor without losing sleep over it. You will want to maintain a very high confidence level about your code. That's why I think unit testing is not optional but should be compulsory for programming in any language, which we'll explain a bit more in the next section.

AN ABSOLUTE NECESSITY FOR BUILDING CONFIDENCE

You'll probably want to know early in the development process if your code works. Deferring finding errors only decreases the accuracy of your project which is probably the opposite of what you want to achieve. This early feedback is beneficial to the quality of your program because you can fix problems while the code and structures are still fresh in your head as opposed to 2 months later.

Since Ruby is a dynamically typed language it defers checking for type errors to as late as possible, which means you would have to wait until runtime to find out about them. This is a less than ideal situation to be in. Enter unit tests; this device is put into being to help you verify the correctness of your code. By writing a test before you write the code you know exactly what your code should do so it becomes easier to write the code that satisfies the test.

Unit tests also help you to find any silly errors you've put in your code in your haste to code stuff up. It functions as a kind of barometer on how good the health of your code is. It helps you build confidence in the code you wrote.

Because I want to be confident that the code I write works as I would expect it, I will use unit tests to drive my process for developing the sample application that accompanies this book.

A BLESSING DURING REFACTORING

Unit testing is an absolute blessing during refactoring. We think we can all agree that if there is one constant in the IT industry that constant is change. So applications generally need to be refactored on a regular basis, this is

usually a pretty dangerous and tedious process after which the application needs to undergo some form of regression testing to find out if you have broken anything else throughout that process.

If you have a full set of unit tests you can immediately locate the problem areas by running your test suite. This helps you eliminate a lot of the problems before you submit your application for regression testing by the QA team. Just for this reason alone we're convinced that unit testing is an absolute must when developing in any language. Why don't we have a look at which tools there are for testing in Ruby?

So how do I go about that then?

For the Ruby language there is a library that is called `Test::Unit` and it provides you a couple of facilities to develop and execute tests. These facilities are that it enables you to express individual tests, it provides you with a couple of ways of executing those tests and it helps you structure your tests.

We want to illustrate this by extending our previous hello world and encapsulating that in a class (Listing 1.13). To do this we first have to create a file that will contain our unit test and add a reference to the unit test library by adding `require 'test/unit'` at the top of the file (Listing 1.12). And next we have to make our class inherit from `Test::Unit::TestCase` to give us access to all the test methods. In ruby the convention is to prefix your test method with `test_`, that way the unit test runner knows which methods are test methods. We are going to be moving hello world into a class that has one public method `print` that will return a string value "Hello, World!!!"

Listing 1.12: Unit testing hello world

```
require 'test/unit'
require 'hello_world'

class TestHelloWorld < Test::Unit::TestCase

  def test_print
    hw = HelloWorld.new
    assert_equal 'Hello, World!!!', hw.print, "The strings should be equal"
  end

end
```

If we try to run that test by invoking the command `rbx test_hello_world.rb` then that test will fail on the line `hw = HelloWorld.new` because that class doesn't exist yet. It may also fail on the line `require 'hello_world'` because that file doesn't exist in the directory where you created that test. To fix those problems we're going to create a file `hello_world.rb` in the folder where you saved `test_hello_world.rb` And the contents of that file looks like listing 1.13

Listing 1.13: The hello world class

```
class HelloWorld

  def initialize
    @message = "Hello, World!!!"
  end

  def print
    @message
  end

end
```

Running the unit test produces the following output:

```
Loaded suite test_hello_world
Started
.
Finished in 0.001 seconds.
```

This section should give you a rudimentary idea of what unit testing is about in Ruby. I can't stress enough how important it is to have proper unit tests. It's a real bonus that unit testing has been made so easy in Ruby. Unit testing is also a really good way to build up a library of what you know by adding a contrived unit test to that

library every time you learn something new. The next section will deal with the tight coupling to types in the classic .NET languages and why you don't need that in Ruby.

1.4.2 Living dangerously: look ma, no interfaces

This section deals with interfaces and the widely adopted interface based design that exists in the C# and Java world amongst others. We'll first discuss what interfaces are and how they are used so that you better grasp which problem they are trying to solve and then we'll debate why they are not necessary in Ruby.

WHAT IS AN INTERFACE EXACTLY?

The C# language only supports single inheritance, which means that any class can only have one parent class. But this limits you in grouping functionality together so C# supports the notion of interfaces, to get around the lack of multiple inheritance. An interface is a reference type without its implementation; it defines the public behavior of a section of a class or a complete class.

One very common analogy to make that illustrates the whole concept is with driving a car. Imagine that if instead of learning how to drive a car, you would have to learn to drive each different type of car you would ever get into. It would be a lot more difficult to change from that trusty old Civic to that shiny new BMW because you would have to learn how to drive it all over. It's a lot easier to just learn how to use the interface of the car: steering wheel, brake, turn signals and gas pedal. That way we don't have to care how the car implements that interface because the interface describes the basic car contract.

WHICH PROBLEMS DO THEY SOLVE?

There are a couple of problems that are being solved by using an interface. These are the four main ones in my opinion.

The first one is that classes may only directly inherit from one base class, but they can implement several interfaces. For an object that needs to exhibit several different facets, this is important. For example an object might represent itself as a printable object through some `IPrintable` interface and as a persisted object through some `IPersistable` interface. Other objects might be `IPrintable` but not `IPersistable` or vice versa. Thus interfaces are a substitute for multiple inheritance.

The second reason is that there can be many implementations of an interface. In the .NET framework collections generally implement the `IEnumerable` interface which has one method `GetEnumerator`, which returns an object that implements the `IEnumerator` interface. This object can iterate (enumerate) over objects in the collection. But of course how it does so will depend on how the collection is structured. Nevertheless, a client that simply needs to iterate over a collection only needs to specify (make a contract that) the collections supports `IEnumerable` (and indirectly `IEnumerator` through its `GetEnumerator` method). It does not need to know anything about how the collection is implemented.

As third usage we could say that interfaces can be used as a publish-and-subscribe mechanism. An object can publish a set of events via an interface and allow an object that wants to subscribe to those events to implement that interface and register itself with the publisher object. Again, the publisher object will probably not know anything about the subscriber object beyond that it implements the interface.

And as last reason we're presenting that the signatures of interfaces (i.e. what declarations they contain) are less likely to change than the signatures of the implementing classes, thus leading to more stable interactions between subsystems across the interface boundary instead of classes directly talking to each other.

So in summary the problems solved by interfaces are:

1. Interfaces allow a work-around for the absence of multiple inheritances in the .NET framework.
2. There may be several implementations of an interface. If a class was used, the several implementations would all have to inherit from this base class, this may be inappropriate (component based).
3. Interfaces allow the specification of a contract when the implementation of that contract is not known, e.g. the publisher specifying the requirements of the subscriber (contract first design).
4. Interfaces are likely to be more stable than their implementing classes and thus tend to isolate the effects of change (loosely coupled).

All of the reasons mentioned in the above text are considered to be good design practices when designing applications. Let's look at an example of what's described above in C#.

AN EXAMPLE IN C#

Suppose we have a class `HelloWorld` that has a method `Print()` and we also have a class `Book` that implements that method `Print()`. Now suppose we want a method `Output` in another class that calls the method `Print()` on both those classes. In that case we could implement an interface on both classes that could be called `IPrintable`. That interface would define the `Print()` method contract. Below you will find the code listings for `HelloWorld` (listing 1.15), `Book` (listing 1.16), `IPrintable` (listing 1.14) and the `Output` method (listing 1.17). The code doesn't do much useful it's just to illustrate how interfaces are being used.

Listing 1.14: The interface `IPrintable`

```
namespace CSharp
{
    public interface IPrintable
    {
        string Message{ get;}
        string Print();
    }
}
```

The interface above defines a public read only property `Message` and a method `Print` that returns a string. The interface only is a contract for how we expect a class to look. Let's implement that class. Below you'll find 2 classes that implement the `IPrintable` interface.

Listing 1.15: The `HelloWorld` class

```
namespace CSharp
{
    public class HelloWorld : IPrintable
    {
        private readonly string message;

        public HelloWorld()
        {
            message = "Hello, World!!!";
        }

        public string Message
        {
            get { return message; }
        }

        public string Print()
        {
            return message;
        }
    }
}
```

The `HelloWorld` class is the most straight forward implementation of the `IPrintable` interface and will always return *"Hello, world!!!"* Next there is the `Book` class which provides another implementation of `IPrintable`, this class allows for some manipulation of the message that will be printed. Both classes are different but they implement the same contract/interface.

Listing 1.16: The `Book` class

```
namespace CSharp
{
    public class Book : IPrintable
    {
        private readonly string message;

        // The user of this class can define the message that will be printed
    }
}
```

```

public Book(string message)
{
    this.message = message;
}

public string Message
{
    get { return message; }
}

// This implementation prefixes the message with Book:
public string Print()
{
    return string.Format("Book: {0}", message);
}
}

```

All that's left for us to do at this point is make sure we can actually see some output of the code we just wrote. For this we need to implement the Main method in a console application.

Listing 1.17: Program.cs with the method Output

```

static void Main(string[] args)
{
    List<object> printables = new List<object>{ new HelloWorld(), new Book("IronRuby In
Action") };

    printables.ForEach(printable => Output(printable as IPrintable));

    object hw = printables[0];
    Console.WriteLine("hw does {0}implement IPrintable", (hw is IPrintable) ? string.Empty :
"not ");
}

static void Output(IPrintable toOutput)
{
    Console.WriteLine(toOutput.Print());
}

/* Generates the following output
PS C:\CSharp\bin\Debug> .\CSharp.exe
Hello, World!!!
Book: IronRuby In Action
hw does implement IPrintable
*/

```

The code listing above is almost the equivalent of the Ruby HelloWorld class we created in our previous section about unit testing. In the next paragraph we'll extend our Ruby program to have a similar main method and a similar output method in our discussion on why interfaces are not needed in a duck typed language.

AND WHY DON'T I NEED INTERFACES NOW?

In the previous paragraphs we've seen how the classic .NET languages are tightly coupled to types. A class and a type are virtually the same and a class can only have one parent. For that reason and a couple of other reasons they have the notion of interfaces. In the Ruby language we are embracing duck typing. If you have a background of programming C#, VB.NET, Java you may feel the temptation to start using Ruby as a statically typed language. We'll explain this by presenting the code for HelloWorld from the previous paragraph with the interface in Ruby. We'll also show you what the correct ways are of using duck typing.

An important insight in the discussion that will follow is to remember that duck typing doesn't really care about the type of an object; it cares more about the behavior that that object has. Consider the classes in listing 1.18.

Listing 1.18: The module Printable, to avoid duplication

```

module Printable
    attr_reader :message

```

```

def print
  @message
end
end

```

We have a module that we can include in other classes and modules so that we don't have code duplication. Analogue to our C# example we're going to create 2 classes that use the module we just specified.

Listing 1.19: The HelloWorld class

```

class HelloWorld
  include Printable

  def initialize
    @message = "Hello, World!!!"
  end
end

```

In listing 1.19 we have the same implementation as with our C# HelloWorld class, it should always output *Hello, World!!!* Below (listing 1.20) we're looking at overriding one of the module's methods and getting the same result as the Book class from our previous example.

Listing 1.20: The Book class

```

class Book
  include Printable

  def initialize(message)
    @message = message
  end

  def print
    "Book: #{@message}"
  end
end

```

Now that we have all the building blocks in place, let's take a look at actually using those blocks (listing 1.21) to provide some output

Listing 1.21: Actually using the classes

```

require 'hello_world'
require 'book'

printables = [HelloWorld.new, Book.new("IronRuby In Action")]

printables.each do |printable|
  puts printable.print
end

hw = printables[0]
puts "hw does #{'not ' unless hw.is_a? Printable}implement Printable"

```

People that come from a statically typed background may have the tendency to write the iteration over the array in a more statically typed fashion (listing 1.22).

Listing 1.22: Making Ruby act statically type, rigidly bound to types

```

printables.each do |printable|
  #the wrong way to do it is to use Ruby as a statically typed language
  fail TypeError.new("#{printable.class} doesn't implement Printable") unless

```

```
printable.is_a? Printable
  puts printable.print
end
```

When you would embrace duck typing you would write a simpler iteration over the array (listing 1.23). Why can you do that with confidence? Because when you call the method `print` on the `printable` instance and the method doesn't exist the program will throw an error anyway, just like what you are doing when you're checking the type. And without that type check your method becomes a lot more flexible because now you could add another type to the mix that doesn't implement the module `Printable` but does respond to the method `print`

Listing 1.23: Embracing duck typing

```
printables.each do |printable|
  # when you don't really care about what's going to happen next.
  # If there is a print method it will execute otherwise it will fail
  puts printable.print
end
```

Sometimes there are these occasions where you do care which behavior is implemented and you want to respond to those things accordingly. This can be done (listing 1.24) but in this case it's important to note that we are checking some behavior on the object not the type or its modules.

Listing 1.24: Checking the behavior of an object

```
printables.each do |printable|
  #if you absolutely need to be sure it will behave correctly
  unless printable.respond_to?(:print)
    fail NoMethodError.new("We expect the method print to be on the object
    <<printable>>")
  end

  puts printable.print
end
```

But before you start walking this path (listing 1.24) you probably should have a real good think about whether it's absolutely necessary to have that check there. Ruby programming is all about simplicity, and the more code you write the less maintainable your code becomes and the more complex it becomes. So make sure that every line is there with a specific purpose.

This concludes our first chapter. The next section summarizes what we've learned so far and will explain what the next steps will entail.

1.5 Summary

We'd like to take this opportunity to explain why Ruby's typing system is a good thing and why you can live without static typing. This is in no way criticism on C# but we need to compare the two systems to show you the contrasts between them.

The static type system is in the mainstream .NET languages don't really help in terms of program security. If C#'s type system were reliable, it wouldn't implement an `InvalidCastException`. But that exception is necessary because there still is some runtime uncertainty in C#. Static typing can be good for optimizing code, and it certainly helps IDEs to deliver stuff like intellisense or autocompletion. It may also help to write refactoring tools for the IDE. However I am yet to be convinced that it guarantees more reliable code.

If you use ASP.NET you use untyped code all the time. Every time you put something in session and you get it out again it gives you an object back that you then cast it to the type you expect it to be. And yet you probably almost never saw an `InvalidCastException`. That's because you structured your code in such a way that it didn't permit that. That is the same in IronRuby. It's very likely that when you use a variable for some purpose, that you will use it a couple of lines later for the same purpose.

As if that wasn't enough, most rubyists tend to program with a TDD or BDD approach to programming which results in lots of short methods and they will have tests for them before they create the method. The short methods mean that the variable only has a very narrow scope. The chance that anything goes wrong with the type is pretty slim. And the testing catches any obvious errors when they happen. Typos usually don't get you very far.

Type-safe isn't that safe after all. That coding in a language like IronRuby is not only safe but it also makes you more productive. You'll probably find that the lack of static typing in IronRuby is an advantage rather than a risk. Once you "get it" you can use it in all its glory and gain huge productivity boosts. This brings us to the next chapter in which we'll look closer at the Ruby language.