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 languages
* REPL : Read Evaluate Print Loop
* The beauty of unit testing
* Duck typing

The first part of this book will help you understand what's in the rest of the book. The first chapter is about some of the specific characteristics of a dynamic. It should also 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.

First I’d like to talk to you about the components of what makes a dynamic language and what makes a language static. Another building block is that Ruby is an interpreted language and is different from a compiled language. The example that will be used throughout this chapter is a hello world implementation, I chose hello world because that is to understand.

After the typing systems have been clarified I’ll highlight some of the most remarkable differences between dynamic typing and static typing. We'll discuss why there are no interfaces in ruby and why you don't really need interfaces.

Another topic of discussion will be REPL. REPL is a nice approach to programming; this should give you a primer on what REPL is. Many languages provide mechanisms for runtime code generation; we'll contrast the complex Reflection.Emit API in our static language with a much simpler mechanism of string evaluation used in the Ruby language.

With all that dynamism going, I would like some reassurance that my code works and that takes us to the importance of unit testing. We’ll see how that can be done using IronRuby using the tools from the Ruby community.

1.1 Setting some ground rules

In this first section I would like to provide some clarification for some of the vocabulary that will be used in this chapter as well as give you my view on why I think you should learn new languages on a regular basis. I will plea in favor of learning a dynamic language but I do believe that all language types are beneficial to your evolution as a programmer.

For example I would give the advice to a novice programmer to start by learning a statically typed and compiled language, such as C#, because those languages force you to take a lot of care and are less forgiving about any mistakes you are making. That should give you a much better basis to learn other types of languages from.

1.1.1 Clarification

In this chapter I will use terms like CLR, DLR, Ruby, IronRuby, TDD and BDD. In this section I'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.

I'm assuming that you know what the .NET framework is. The Common Language Runtime (CLR) is a part of the .NET framework and indicates the virtual machine on which Common Intermediate Language (CIL/IL) bytecode is run. The CIL is the lowest-level human-readable programming language in the .NET Framework.

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. I will use IronRuby and Ruby interchangeably throughout this book but I will always be talking about IronRuby in the context of this book.

The last terms I 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 which aid you in maintaining your application and give you a huge confident boost about your code. 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 it's predecessor.

In the next part of this section I'd like share my views on why learning a different language is a good thing. This paragraph will give you some of my 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 those debates because I think those are futile and serve no purpose but to antagonize. 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.

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 C# I loved the introduction of generics but they weren't always able to give me the flexibility that I needed. I wanted more 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 of him 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 2 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 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 is 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 keystrokes, 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.

I would like to learn a new programming language every year. 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 your program to run as fast as possible and then you would probably to use C++ for that application. However if speed doesn't matter that much you can use a dynamic language, 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 I'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 I can get to some coding, I 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 my 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 I’ll be using to demonstrate my 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 I'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 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 some optimizations may have been made by the compiler. 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.

An interpreted language means that the code you feed it is 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 and as such, it 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 a lot of information about its run-time environment 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 they 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.

A nice analogy to make can be that 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 translates a task on the spot to your business partner. Your business partner would then go and complete that task and wait for you to give him a next task.

In the compiled scenario you would contact a translator beforehand and have him translate all the tasks. You would then meet with your business partner and hand him over the list of tasks. He will then go do the jobs until the list of tasks is finished.

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

1.2.2 Typing

1.2.2.1 Static, dynamic typing and type inference

The process of declaring the variables and function return values is called static typing. A static 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 casted or converted to another type. The classic .NET languages are all statically typed and that is enforced by the compiler. 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 to write a program.

I'll illustrate this point with the example below. It shows you some of the most basic things you can do with a statically typed language.

Static typing in c#:

int i = 5;

i += 5;

//This is valid because console writeline has an appropriate overload defined.

System.Console.WriteLine(i); //🡺 10

// i = "hello"; // won't compile because we labeled i to be an int.

// To convert this value to a string we can rely on the limited type coercion that the C# compiler understands.

// or we can convert it explicitly

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

The languages built on top of the DLR (I'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 declarations before they are used. You could also make a distinction between the two typing methods by when the type checking occurs; this is more accurate in my opinion because both typing mechanisms have casting and conversion techniques. This distinction would then be that static typed languages perform their type checking at compile time where dynamic typed languages defer 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# 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.

Type inference in C# 3.0

var i = 5;

i = i + 5; //This is valid because type could be inferred from assignment

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

// i = "hello"; // Still won't compile because of the inferred type of i from line 1

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

Static and dynamic typing is different from strong and weak typing. People use these terms interchangeably but they are 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.

Dynamically typed in IronRuby

#dynamic but strong typed.

a = 5

a = 5 + 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.to\_s => "10"

# The example below is a valid use in dynamic languages

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

The next parts of this section will deal with weak and strong typing as well as explain what duck typing is all about. The distinction between strong typing and weak typing is an important concept to grasp because that is what makes a world of difference and is often a big source of confusion when people talk about programming languages.

1.2.2.2 Strong and weak typing

A strong 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 question 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 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 IronRuby are strong typed.

1.2.2.3 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's close enough." 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.

1.2.3 Why ruby's type system is a good thing

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.

Ruby makes more than up for it by representing a kind of best of breed implementation of a programming language. Ruby borrows 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.

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

After that theoretical discussion I'm 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 I'll talk you through the steps of getting the source and compiling it.

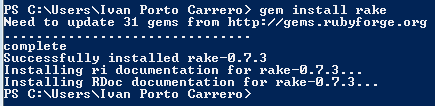
After that we'll get to do a little bit of code using the interactive console, at which time I'll introduce you to the concept of REPL and why the console will be your new best programming buddy. When you're familiar with the console it's time to look at what it means to have a first class eval method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

1.3.1 Bootstrapping your environment

Right let's get down to business. In this chapter I'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 I chose for the option to let rake build it. So we'll also be getting the C-Ruby implementation.

For the C-Ruby version you can get it from <http://www.ruby-lang.org/en/downloads/>. I downloaded the one click installer and when the install completed. I added C:\Ruby\bin to my PATH environment variable. Next I typed gem install rake in a newly opened console window.

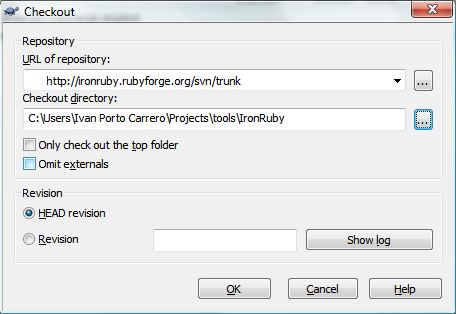
The PowerShell window with the gem command



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

The source code for IronRuby can be downloaded from [rubyforge](http://rubyforge.org/projects/ironruby). To download the source control you need a subversion client. My client of choice is TortoiseSVN. 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 following window.

The tortoise check out window



At this moment we 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 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.

Compiling the debug version of ironruby with rake



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

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 you're 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. I'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.

1.3.2.1 REPL

REPL stands for Read-Evaluate-Print-Loop and signifies the repeated process that you can employ to develop in Ruby. In fact REPL often manifests itself as a type of interactive console in which you type a line of code.

That line of code will be read by the console then evaluated and the result will be printed in the console after which the console will loop to read the next statement you feed it.

The moment I got 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, experimentation, and debugging code, don't you agree? IronRuby does REPL, why don't we try it out?

1.3.2.2 The console, your new best friend

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 piecewise, 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 an implementation we are going to use the interactive console to test it. So let's spin up 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 into. Type ./rbx to start up the console.

Starting up 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. Ooh that seems to have done something. Well that would be correct but nothing has been executed. The sign => is a way for the console to tell you the result of the last expression that has been validated by it.

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"

>>>

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 (System.Console.WriteLine). That should be the complete implementation of Hello world in the console.

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 !!!

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 I 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 is more than one way to run ruby programs and that is what we'll be viewing next.

1.3.2.3 More ways to run ruby programs

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 before. 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 following output.

The output of running the helloworld program.

PS C:\IronRuby\build\debug> rbx helloworld.rb

Hello world !!!

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 next discussion I 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 then 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 I'm 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. I 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. I'll turn to the good old hello world program so that it is simple enough for everyone to understand.

1.3.3.1 Reflection.Emit: executing arbitrary code

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 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 implement our hello world implementation in C# we would do have to write something like this:

Hello world in C#

public static void Main(string[] args)

{

System.Console.WriteLine("hello world");

}

And when we compiled that bit of code it could be a bytecode representation of:

Hello world in IL

.method public static void MyMain() cil managed

{

.entrypoint

ldstr "hello world!"

call void [mscorlib]System.Console::WriteLine(string)

ret

}

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

Generating code at runtime with C#

DynamicMethod dm = new DynamicMethod("HelloWorld", typeof(void), new Type[] { }, typeof(HelloWorldLCG), false); //Get a reference to the dynamic method

ILGenerator il = dm.GetILGenerator(); //Get the IL generator

il.Emit(OpCodes.Ldstr, "hello, world"); // Load a string onto the heap

il.Emit(OpCodes.Call, typeof(Console).GetMethod("WriteLine", new Type[] { typeof(string) })); //Call Console.Writeline

il.Emit(OpCodes.Ret); // return from the method

dm.Invoke(null, null); // invoke the dynamic method

The point I'm 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. Next we'll look at how to achieve this in IronRuby.

1.3.3.2 Eval: executing arbitrary code

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 I'm just going to show that it works I'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 at 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 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"') in our console to see that result.

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 implications, definitely if you're coming here from languages like C#, VB.NET, Java, C++… and that is exactly what the next section will be dealing with. We'll look at some of the implications from working with a dynamic language.

1.4 And some implications

I 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, to prove that your code behaves as expected, is to write unit tests to prove that your code works 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 I thought I'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 inheritance. A term often used is interfaced 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.

It 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.

1.4.1.1 An absolute necessity

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.

1.4.1.2 A blessing during refactoring

Unit testing is an absolute blessing during refactoring. I think we 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 haven't 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 I am 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?

1.4.1.3 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.

I want to illustrate this by extending our previous hello world and encapsulating that in a class. To do this I first have to create a file that will contain my unit test and add a reference to the unit test library by adding require 'test/unit' at the top of the file. And next I have to make my class inherit from Test::Unit to give me access to all the 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 !!!". How would that test class then look?

Code listing of test\_hello\_world.rb

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 so:

Code listing for hello\_world.rb

class HelloWorld

def initialize

@message = "Hello world !!!"

end

def print

@message

end

end

When we now try to run that test it will produce the following output which is what we are after:

Output of running the unit test:

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.

1.4.2.1 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 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.

1.4.2.2 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 I'm 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 inheritance 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.

1.4.2.3 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, IPrintable and the Output method. The code doesn't do much useful it's just to illustrate how interfaces are being used.

Code listing for the interface IPrintable

namespace CSharp

{

public interface IPrintable

{

string Message{ get;}

string Print();

}

}

Code listing for HelloWorld.cs

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;

}

}

}

Code listing for Book.cs

namespace CSharp

{

public class Book : IPrintable

{

private readonly string message;

public Book(string message)

{

this.message = message;

}

public string Message

{

get { return message; }

}

public string Print()

{

return message;

}

}

}

Code listing for Program.cs with the method Output

static void Main(string[] args)

{

List<object> printables = new List<object>{ new HelloWorld(), new Book("IronPython 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 !!!

IronPython 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.

1.4.2.4 And why don't I need them?

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. I'll explain this by presenting the code for HelloWorld from the previous paragraph with the interface in ruby. I'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 following classes:

module Printable

def print

@message

end

end

class HelloWorld

include Printable

def initialize

@message = "Hello world !!!"

end

end

class Book

include Printable

def initialize(message)

@message = message

end

end

printables = [HelloWorld.new, Book.new("IronPython 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 the following style:

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. 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 that does respond to the method print

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 but in this case it's important to note that we are checking some behavior on the object not the type or it's modules.

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 TypeError.new("We expect the method print to be on the object <<printable>>")

puts printable.print

end

But before you start walking this path 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

I'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 I need to compare the two systems to show you the contrasts between them.

It's a fact that the static type system 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 do clever things with tooltip help. However I am yet to be convinced that it guarantees more reliable code.

When you'll be using IronRuby for a while, you'll realize that dynamically typed languages actually are more of a blessing than a curse. They really increase your productivity. It may even surprise you that you'll find to be proven wrong about the imminent mayhem caused by having no compiler. Why is it that there is no mayhem and the world doesn't stop spinning?

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. Typo's etc usually don't get very far.

Which leads me to believe that type-safe isn't that safe after all, and 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.