The Book Of Ruby

Huw Collingbourne

Author's web site: http://www.sapphiresteel.com/

ABOUT THE AUTHOR

Huw Collingbourne is Technology Director of SapphireSteel Software (*http://www.sapphiresteel.com/*), developers of the '**Ruby In Steel**' Ruby and Rails IDE for Visual Studio and the '**Amethyst**' IDE for Adobe Flex. Huw is a well known technology writer in the UK and has written numerous opinion and programming columns, including tutorials on C#, Delphi, Java, Smalltalk and Ruby for a number of computer magazines such as Computer Shopper, PC Pro and PC Plus. He is author of the free ebook, The Little Book Of Ruby, and is the editor the online computing magazine, Bitwise (*www.bitwisemag.com*). Huw has an MA in English from the University of Cambridge.

INTRODUCTION

Getting Started With Ruby

As you are now reading a book on Ruby, I think it is safe to work on the assumption that you don't need me to persuade you of the merits of the Ruby language. Instead I'll take the somewhat unconventional step of starting with a warning: many people are attracted to Ruby by its simple syntax and its ease of use. They are wrong. Ruby's syntax may look simple at first sight but, the more you get to know the language the more you will realize that it is, on the contrary, extremely complex. The plain fact of the matter is that Ruby has a number of pitfalls just waiting for unwary programmers to drop into.

In this book it is my aim to guide you safely over the pitfalls and lead you through the choppy waters of Ruby's syntax and class libraries. In the process, I'll be exploring both the smooth, well-paved highways and the gnarlier, bumpy little byways of Ruby. By the end of the journey, you should be able to use Ruby safely and effectively without getting caught out by any unexpected hazards along the way.

The Book Of Ruby concentrates principally on version 1.8.x of the Ruby language. While a version 1.9 of Ruby has been released, Ruby 1.8 is still far more widely used. Ruby 1.9 may be regarded as a stepping stone towards Ruby 2.0. In most respects the syntax of Ruby 1.9 is close to that of Ruby 1.8 but you should be aware that there are some differences and complete compatibility is not guaranteed.

HOW TO READ THIS BOOK

The book is divided up into bite-sized chunks. Each chapter introduces a theme which is subdivided into sub-topics. Each programming topic is accompanied by one or more small self-contained, ready-to-run Ruby programs.

If you want to follow a well-structured 'course', read each chapter in sequence. If you prefer a more hands-on approach, you can run the programs first and refer to the text when you need an explanation. If you already have some experience of Ruby, feel free to cherry-pick topics in any order that you find useful. There are no monolithic applications in this book so you don't have to worry that you might 'lose the plot' if you read the chapters out of order!

DIGGING DEEPER

Every chapter apart from the first includes a section called 'Digging Deeper'. This is where we explore specific aspects of Ruby (including a few of those gnarly byways I mentioned a moment ago) in greater depth. In many cases you could skip the Digging Deeper sections and still learn all the Ruby you will ever need. On the other hand, it is in the Digging Deeper sections that we often get closest to the inner workings of Ruby so, if you skip them, you are going to miss out on some pretty interesting stuff.

MAKING SENSE OF THE TEXT

In The Book Of Ruby, any Ruby source code is written like this:

```
def saysomething
  puts( "Hello" )
end
```

When there is a sample program to accompany the code, the program name is shown in a box on the right-hand side of the page, like this:

helloname.rb

Explanatory notes (which generally provide some hints or give a more in-depth explanation of some point mentioned in the text) are shown in a box like this:

> **This is an explanatory note**. You can skip it if you like – but if you do so, you may miss something of interest…!

Ruby and Rails

WHAT IS RUBY?

Ruby is a cross-platform interpreted language which has many features in common with other 'scripting' languages such as Perl and Python. It has an 'English language' style syntax which looks somewhat Pascal-like at first sight. It is thoroughly object oriented, and has a good deal in common with the great-granddaddy of 'pure' OO languages, Smalltalk. It has been said that the languages which most influenced the development of Ruby were: Perl, Smalltalk, Eiffel, Ada and Lisp. The Ruby language was created by Yukihiro Matsumoto (commonly known as 'Matz') and it was first released in 1995.

WHAT IS RAILS?

Currently much of the excitement surrounding Ruby can be attributed to a web development framework called Rails – popularly known as 'Ruby On Rails'. Rails is an impressive framework but it is not the be-all and end-all of Ruby. Indeed, if you were to leap right into Rails development without first mastering Ruby, you might find that you end up creating applications that you don't even understand (this is, in fact, all too common among Ruby On Rails novices). Understanding Ruby is a necessary prerequisite of understanding Rails.

DOWNLOAD RUBY

You can download the latest version of Ruby from **http://www.ruby-lang.org**. Be sure to download the binaries (not merely the source code). On a PC you can install Ruby using the Ruby Installer for Windows:

http://rubyinstaller.rubyforge.org/wiki/wiki.pl

Alternatively, if you are using the Ruby In Steel IDE, you can install Ruby, Rails, Ruby In Steel and all the other tools you will need using the Ruby In Steel 'All-in-one installer' available on the site's Download page:

http://www.sapphiresteel.com/

GET THE SOURCE CODE OF THE SAMPLE PROGRAMS

All the programs in every chapter in this book are available for download as a Zip archive from http://www.sapphiresteel.com/The-Book-Of-Ruby. When you unzip the programs you will find that they are grouped into a set of directories – one for each chapter. For the benefit of programmers using Ruby In Steel (the Visual Studio IDE developed by the author of this book's company), you will be able to load the programs as Visual Studio solutions into Ruby In Steel For Visual Studio 2008, with the programs for each chapter arranged on the branches of a tree in the Project Manager. If you are using another editor or IDE, load each Ruby program, one by one, as it is needed. Users of Ruby In Steel for Visual Studio 2005 may import or convert the projects (via the *File New/Open* menu).

RUNNING RUBY PROGRAMS

It is often useful to keep a command window open in the source directory containing your Ruby program files. Assuming that the Ruby interpreter is correctly pathed on your system, you will then be able to run programs by entering `ruby <program name>` like this:

```
ruby 1helloworld.rb
```

If you are using Ruby In Steel you can run the programs in the interactive console by pressing CTRL+F5 or run them in the debugger by pressing F5.

THE RUBY LIBRARY DOCUMENTATION

The Book Of Ruby covers many of the classes and methods in the standard Ruby library - but by no means all of them! At some stage, therefore, you will need to refer to documentation on the full range of classes used by Ruby. Fortunately, the Ruby class library contains embedded documentation which has been extracted and compiled into an easily browsable reference which is available in several formats. For example, refer to this online documentation which is shown in a multi-pane web page:

http://www.ruby-doc.org/core/

Alternatively, here you can browse the library alphabetically:

http://www.ruby-doc.org/stdlib/

The above page contains instructions for downloading the documentation for offline browsing. There is also a page from which the library (and other) documentation may be downloaded in various formats, versions and languages:

http://www.ruby-doc.org/downloads

OK, that's enough of the preamble – let's get down to work. Time to move straight on to Chapter One…

CHAPTER ONE

Strings, Numbers, Classes and Objects

The first thing to know about the Ruby language is that it's easy to use. To prove this, let's look at the code of the traditional 'Hello world' program. Here it is:

1helloworld.rb

```
puts 'hello world'
```

That's it in its entirety. One method, puts, and one string, 'hello world'. No headers or class definitions, no import sections or 'main' functions. This really is as simple as it gets. Load up the code, **1helloworld.rb**, and try it out.

GETTING AND PUTTING INPUT

Having 'put' a string to the output (here, a command window), the obvious next step is to 'get' a string. As you might guess, the Ruby method for this is gets. The **2helloname.rb** prompts the user for his or her name – let's suppose it's "Fred" - and then displays a greeting: "Hello Fred". Here is the code:

2helloname.rb

```
print( 'Enter your name: ' )
name = gets()
puts( "Hello #{name}" )
```

While this is still very simple, there are a few important details that need to be explained. First, notice that I've used print rather than puts to display the prompt. This is because puts adds a linefeed at the end whereas print does not; in the present case I want the cursor to remain on the same line as the prompt.

On the next line I use gets() to read in a string when the user presses Enter. This string is assigned to the variable, name. I have not pre-declared this variable, nor have I specified its type. In Ruby you can create variables as and when you need them and Ruby 'infers' their types. In the present case I have assigned a string to name so Ruby knows that the type of the name variable must be a string.

Note: **Ruby is case sensitive**. A variable called myvar is different from one called myVar. A variable such as name in our sample project must begin with a lowercase character (if it begins with an uppercase character Ruby will treat it as a constant – I'll have more to say on constants in a later chapter).

Incidentally, the brackets following gets() are optional as are the brackets enclosing the strings after print and puts; the code would run just the same if you removed the brackets. However, brackets can help to resolve ambiguities and, in some cases, the interpreter will warn you if you omit them.

STRINGS AND EMBEDDED EVALUATION

The last line in our sample code is rather interesting:

```
puts( "Hello #{name}" )
```

Here the name variable is embedded into the string itself. This is done by placing the variable between two curly braces preceded by a hash ('pound') character #{}. This kind of 'embedded' evaluation only works with strings delimited by double quotes. If you were to try this with a string delimited by single quotes, the variable would not be evaluated and the string 'Hello #{name}' would be displayed exactly as entered.

It isn't only variables which can be embedded in double-quoted strings. You can also embed non-printing characters such as newlines "\n" and tabs "\t". You can even embed bits of program code and mathematical expressions. Let's assume that you have a method called showname, which returns the string 'Fred'.

The following string would, in the process of evaluation, call the showname method and, as a result, it would display the string "Hello Fred":

```
puts "Hello #{showname}"
```

See if you can figure out what would be displayed by the following:

3string_eval.rb

```
puts( "\n\t#{(1 + 2) * 3}\nGoodbye" )
```

Now run the **3string_eval.rb** program to see if you were right.

NUMBERS

Numbers are just as easy to use as strings. For example, let's suppose you want to calculate the selling price or 'grand total' of some item based on its ex-tax value or 'subtotal'. To do this you would need to multiply the subtotal by the applicable tax rate and add the result to the value of the subtotal. Assuming the subtotal to be $100 and the tax rate to be 17.5%, this Ruby program does the calculation and displays the result:

4calctax.rb

```
subtotal = 100.00
taxrate = 0.175
tax = subtotal * taxrate
puts "Tax on $#{subtotal} is $#{tax}, so grand total is $#{subtotal+tax}"
```

Obviously, this program would be more useful if it could perform calculations on a variety of subtotals rather than calculating the same value time after time! Here is a simple version of a Calculator that prompts the user to enter a subtotal:

```
taxrate = 0.175
print "Enter price (ex tax): "
s = gets
subtotal = s.to_f
tax = subtotal * taxrate
puts "Tax on $#{subtotal} is $#{tax}, so grand total is $#{subtotal+tax}"
```

Here `s.to_f` is a method of the String class. It attempts to convert the string to a floating point number. For example, the string "145.45" would be converted to the floating point number, 145.45. If the string cannot be converted, 0.0 is returned. So, for instance, `"Hello world".to_f` would return 0.0.

Comments...

Many of the source code examples that comes with this book are documented with comments which are ignored by the Ruby interpreter. A comment may be placed after the pound (or 'hash') character, `#`. The text on a line following this character is all treated as a comment:

```
# this is a comment

puts( "hello" ) # this is also a comment
```

If you wish to comment out multiple lines of text you can place `=begin` at the start and `=end` at the end (both `=begin` and `=end` must be flush with the left margin):

```
=begin
 This is a
 multiline
 comment
=end
```

TESTING A CONDITION: IF … THEN

The problem with the simple tax calculator code shown above is that it accepts minus subtotals and calculates minus tax on them – a situation upon which the Government is unlikely to look favourably! I therefore need to check for minus figures and, when found, set them to zero. This is my new version of the code:

5taxcalculator.rb

```
taxrate = 0.175
print "Enter price (ex tax): "
s = gets
subtotal = s.to_f

if (subtotal < 0.0)  then
  subtotal = 0.0
end

tax = subtotal * taxrate
puts "Tax on $#{subtotal} is $#{tax}, so grand total is $#{subtotal+tax}"
```

The Ruby `if` test is similar to an `if` test in other programming languages. Note, however, that the brackets are once again optional, as is the keyword `then`. However, if you were to write the following, with no line break after the test condition, the `then` would be obligatory:

```
if (subtotal < 0.0) then subtotal = 0.0 end
```

Putting everything on one line like this adds nothing to the clarity of the code, which is why I prefer to avoid it. My long familiarity with Pascal instinctively makes me want to add a `then` after the `if` condition but, as this really is not required, you may look upon this as a wilful eccentricity of mine. The `end` keyword that terminates the `if` block is *not* optional. Forget to add it and your code will not run.

LOCAL AND GLOBAL VARIABLES

In the previous example, I assigned values to variables such as `subtotal`, `tax` and `taxrate`. Variables such as these which begin with a lowercase character are local variables. This means that they only exist within a specific part of a program – in other words, they are restricted to a well-defined scope. Here is an example:

variables.rb

```
localvar = "hello"
$globalvar = "goodbye"

def amethod
  localvar = 10
  puts( localvar )
  puts( $globalvar )
end

def anotherMethod
  localvar = 500
  $globalvar = "bonjour"
  puts( localvar )
  puts( $globalvar )
end
```

Here there are three local variables called localvar. One is assigned the value, "hello world" within the 'main scope' of the program; two others are assigned integers within the scope of two separate methods: since each local variable has a different scope, the assignments have no affect on the other local variables with the same name in different scopes. You can verify this by calling the methods in turn:

```
amethod           #=> localvar = 10
anotherMethod     #=> localvar = 500
amethod           #=> localvar = 10
puts( localvar )  #=> localvar = "hello"
```

On the other hand, a global variable – one that begins with the dollar $ character - has global scope. When an assignment is made to a global variable inside a method, that affects the value of that variable elsewhere in the program too:

```
amethod           #=> $globalvar = "goodbye"
anotherMethod     #=>  $globalvar = "bonjour"
amethod           #=>  $globalvar = "bonjour"
puts( $globalvar ) #=>  $globalvar = "bonjour"
```

CLASSES AND OBJECTS

Instead of going through all the rest of Ruby's syntax - its types, loops, modules and so on (but fear not, we'll come back to those soon) - let's move rapidly on to take a look at how to create classes and objects.

Classes, Objects and Methods

A 'class' is the blueprint for an object. It defines the data an object contains and the way it behaves. Many different objects can be created from a single class. So you might have one Cat **class** but three cat **objects**: *tiddles*, *cuddles* and *flossy*. A **method** is like a function or subroutine that is defined inside the class.

It may seem no big deal to say that Ruby is object oriented. Aren't all languages, these days? Well, up to a point. Most modern 'object oriented' languages (Java, C++, C#, Object Pascal and so on) have a greater or lesser degree of OOP features. Ruby, on the other hand, is obsessively object oriented. In fact, unless you have programmed in Smalltalk or Eiffel (languages which are even more obsessive than Ruby about objects), it is likely to be the most object oriented language you have ever used. Every chunk of data – from simple things like numbers and strings to complicated things like files and modules – is treated as an object. And almost everything you do with an object is done by a method. Even 'operators' such as plus + and minus - are methods. Consider the following:

```
x = 1 + 2
```

Here + is a method of the Fixnum (Integer) object, 1. The value 2 is sent to this method; the result, 3, is returned and this is assigned to the object, x. Incidentally, the operator, =, is one of the rare exceptions to the rule that "everything you do with an object is done by a method". The assignment operator is a special built-in 'thingummy' (this is not the formal terminology, I hasten to add) and it is not a method of anything.

Now let's see how to create objects of our own. As in most other OOP languages, a Ruby object is defined by a class. The class is like a blueprint from which individual objects are constructed. For example, this class defines a dog:

```
class Dog
  def set_name( aName )
    @myname = aName
  end
end
```

Note that the class definition begins with the keyword class (all lowercase) and the name of the class itself, which must begin with an uppercase letter. The class contains a method set_name. This takes an incoming argument, aName. The body of the method assigns the value of aName to a variable called @myname.

INSTANCE VARIABLES

Variables beginning with the @ character are 'instance variables' – that means that they belong to individuals objects – or 'instances' of the class. It is not necessary to pre-declare instance variables. I can create instances of the Dog class (that is, 'dog objects') by calling the new method. Here I am creating two dog objects (note that while class names begin uppercase letters, object names begin with lowercase letters):

```
mydog = Dog.new
yourdog = Dog.new
```

At the moment, these two dogs have no names. So the next thing I do is call the set_name method to give them names:

```
mydog.set_name( 'Fido' )
yourdog.set_name( 'Bonzo' )
```

Having given each dog a name, I need to have some way to find out its name later on. How shall I do this? I can't poke about inside an object to get at the @name variable, since the internal details of each object are known only to the object itself. This is a fundamental principle of 'pure' object orientation: the data inside each object is private. There are precisely defined ways into each object (for example, the method set_name) and precisely defined ways out. Only the object itself can mess around with its internal state. The outside world cannot. This is called 'data hiding' and it is part of the principle of 'encapsulation'.

Encapsulation

In Ruby, encapsulation is not quite as rigorously enforced as it initially appears. There are some very dirty tricks that you can do to mess around inside an object. For the sake of clarity (and to make sure you, and I, don't have nightmares), we shall, for now, silently pass over these features of the language.

Since we need each dog to know its own name, let's provide the Dog class with a `get_name` method:

```
def get_name
  return @myname
end
```

The `return` keyword here is optional. When it is omitted, Ruby methods will return the last expression evaluated.

For the sake of clarity (and to avoid unexpected results from methods of more complexity than this one!) I shall make a habit of explicitly returning any values which I plan to use.

Finally, let's give the dog some behaviour by asking it to talk. Here is the finished class definition:

```
class Dog
  def set_name( aName )
    @myname = aName
  end

  def get_name
    return @myname
  end

  def talk
    return 'woof!'
  end
end
```

Now, we can create a dog, name it, display its name and ask it to talk like this:

```
mydog = Dog.new
mydog.set_name( 'Fido' )
puts(mydog.get_name)
puts(mydog.talk)
```

6dogs.rb

I've written an expanded version of this code in the **6dogs.rb** program. This also contains a Cat class which is similar to the Dog class apart from the fact that its `talk` method, naturally enough, returns a *miaow* instead of a *woof*.

> ***Oops!*** It seems that this program contains an error.
>
> The object named *someotherdog* never has a value assigned to its `@name` variable. Fortunately, Ruby doesn't blow up when we try to display this dog's name. Instead it just prints 'nil'. We'll shortly look at a simple way of making sure that errors like this don't happen again…

MESSAGES, METHODS AND POLYMORPHISM

This example, incidentally, is based on a classic Smalltalk demo program which illustrates how the same 'message' (such as `talk`) can be sent to different objects (such as cats and dogs), and each different object responds differently to the same message with its own special method (here the `talk` method). The ability to have different classes containing methods with the same name goes by the fancy Object Orientated name of 'polymorphism' – a term which, once remembered, can safely be forgotten…

When you run a program such as **6dogs.rb**, the code is executed in sequence. The code of the classes themselves is not executed until instances of those classes (i.e. objects) are created by the code at the bottom of the program. You will see that I frequently mix class definitions with 'free standing' bits of code which executes when the program is run. This may not be the way you would want to write a major application but for just 'trying things out' it is extremely convenient.

Free-standing Bits Of Code...?

If Ruby is really an Object Orientated language, you may think it odd that we can enter 'free floating' methods. In fact, it turns out that when you run a program, Ruby creates a **main** object and any code that appears inside a code unit is, in spite of appearances, not 'free floating' at all; it is, in fact, running inside the **main** object. You can easily verify this. Create a new source file, add the code below then run it to view the output:

```
puts self
puts self.class
```

One obvious defect of my program is that the two classes, Cat and Dog, are highly repetitious. It would make more sense to have one class, Animal, which has `get_name` and `set_name` methods and two descendent classes, Cat and Dog, which contain only the behaviour – woofing or miaowing – specific to that species of animal. We'll be finding out how to do this in the next chapter.

CONSTRUCTORS – NEW AND INITIALIZE

For now, let's take a look at another example of a user-defined class. Load up **7treasure.rb**. This is an adventure game in the making. It contains two classes, Thing and Treasure. The Thing class is very similar to the Cat and Dog classes from the last program – apart from the fact that it doesn't woof or miaow, that is.

The Treasure class hasn't got `get_name` and `set_name` methods. Instead, it contains a method named `initialize` which takes two arguments whose values are assigned to the `@name` and `@description` variables:

7treasure.rb

```
def initialize( aName, aDescription )
  @name        = aName
  @description  = aDescription
end
```

When a class contains a method named initialize this will be automatically called when an object is created using the new method. It is a good idea to use an initialize method to set the values of an object's instance variables.

This has two clear benefits over setting each instance variable using methods such set_name. First of all, a complex class may contain numerous instance variables and you can set the values of all of them with the single initialize method rather than with many separate 'set' methods; secondly, if the variables are all automatically initialised at the time of object creation, you will never end up with an 'empty' variable (like the *nil* value returned when we tried to display the name of someotherdog in the previous program).

Finally, I have created a method called to_s which is intended to return a string representation of a Treasure object. The method name, to_s, is not arbitrary. The same method name is used throughout the standard Ruby object hierarchy.
In fact, the to_s method is defined for the Object class itself which is the ultimate ancestor of all other classes in Ruby. By redefining the to_s method, I have added new behaviour which is more appropriate to the Treasure class than the default method. In other words, I have 'overridden' its to_s method.

The new method creates an object so it can be thought of as the object's 'constructor'. However, you should not normally implement your own version of the new method (this *is* possible but it is generally not advisable). Instead, when you want to perform any 'setup' actions – such as assigning values to an object's internal variables - you should do so in a method named initialize. Ruby executes the initialize method immediately after a new object is created.

Garbage Collection

In many languages such as C++ and Delphi for Win32, it is the programmer's responsibility to destroy any object that has been created when it is no longer required. In other words, objects are given *destructors* as well as *constructors*. In Ruby, you don't have to do this since Ruby has a built-in 'garbage collector' which automatically destroys objects and reclaims the memory they used when they are no longer referenced in your program.

INSPECTING OBJECTS

Incidentally, notice too that I have 'looked inside' the Treasure object, t1, using the inspect method:

```
t1.inspect
```

The inspect method is defined for all Ruby objects. It returns a string containing a human-readable representation of the object. In the present case, it displays something like this:

```
#<Treasure:0x28962f8 @description="an Elvish weapon forged of gold",
@name="Sword">
```

This begins with the class name, Treasure; the name is followed by a number, which may be different from the number shown above – this is Ruby's internal identification code for this particular object; then there are the names and values of the object's variables.

Ruby also provides the p method as a shortcut to inspecting objects and printing out their details, like this:

p.rb

```
p( anobject )
```

To see how to_s can be used with a variety of objects and to test how a Treasure object would be converted to a string in the absence of an overridden to_s method, try out the **8to_s.rb** program.

8to_s.rb

```
puts(Class.to_s)          #=> Class
puts(Object.to_s)         #=> Object
puts(String.to_s)         #=> String
puts(100.to_s)            #=> 100
puts(Treasure.to_s)       #=> Treasure
```

As you will see, classes such as Class, Object, String and Treasure, simply return their names when the `to_s` method is called. An object, such as the Treasure object, `t`, returns its identifier – which is the same identifier returned by the `inspect` method:

```
t = Treasure.new( "Sword", "A lovely Elvish weapon" )
puts(t.to_s)
     #=> #<Treasure:0x3308100>
puts(t.inspect)
     #=> #<Treasure:0x3308100
              @name="Sword", @description="A lovely Elvish weapon">
```

While the **7treasure.rb** program may lay the foundations for a game containing a variety of different types of object, its code is still repetitive. After all, why have a Thing class which contains a name and a Treasure class which also contains a name? It would make more sense to regard a Treasure as a '*type of*' Thing. In a complete game, other objects such as Rooms and Weapons might be yet other 'types of' Thing. It is clearly time to start working on a proper class hierarchy. That's what we shall do in the next chapter…

CHAPTER TWO

Class Hierarchies, Attributes and Class Variables

We ended the last lesson by creating two new classes: a Thing and a Treasure . In spite of the fact that these two classes shared some features (notably both had a 'name'), there was no connection between them.

Now, these two classes are so trivial that this tiny bit of repetition doesn't really matter much. However, when you start writing real programs of some complexity, your classes will frequently contain numerous variables and methods; and you really don't want to keep recoding the same things over and over again.

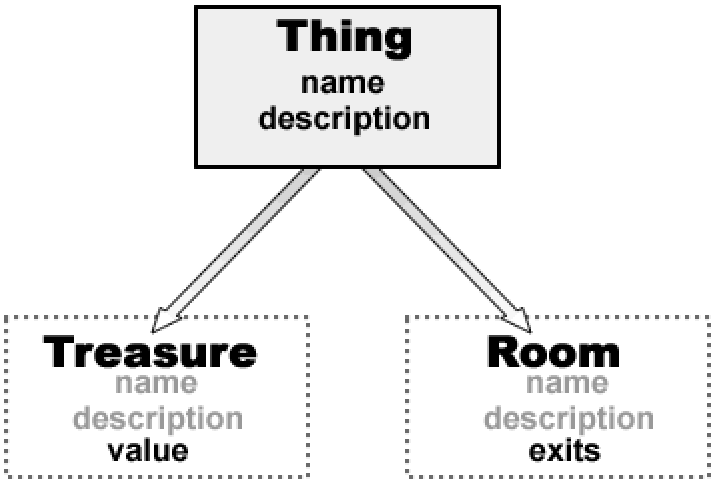
It makes sense to create a class hierarchy in which one class may be a 'special type' of some other ('ancestor') class, in which case it will automatically inherit the features of its ancestor. In our simple adventure game, for instance, a Treasure is a special type of Thing so the Treasure class should inherit the features of the Thing class.

Class Hierarchies – Ancestors and Descendants: In this book, I shall often talk about 'descendant' classes 'inheriting' features from their 'ancestor' classes. These terms deliberately suggest a kind a family relationship between 'related' classes. Each class in Ruby has only one parent. It may, however, descend from a long and distinguished family tree with many generations of parents, grandparents, great-grandparents and so on…

The behaviour of Things in general will be coded in the Thing class itself. The Treasure class will automatically 'inherit' all the features of the Thing class, so we won't need to code them all over again; it will then add some additional features, specific to Treasures.

As a general rule, when creating a class hierarchy, the classes with the most generalised behaviour are higher up the hierarchy than classes with more specialist behaviour. So a Thing class with just a name and a description, would be the ancestor of a Treasure class which has a name, a description and, additionally, a value; the Thing class might also be the ancestor of some other specialist class such as a Room which has a name, a description and also exits – and so on…

One Parent, Many Children...



This diagram shows a Thing class which has a *name* and a *description* (in a Ruby program, these might be internal variables such as `@name` and `@description` plus some methods to access them). The Treasure and Room classes both descend from the Thing class so they automatically 'inherit' a *name* and a *description*. The Treasure class adds one new item: *value* – so it now has *name, description* and *value*; The Room class adds *exits* – so it has *name*, *description* and *exits*.

1adventure.rb

Let's see how to create a descendant class in Ruby. Load up the **1adventure.rb** program. This starts simply enough with the definition of a Thing class which has two instance variables, `@name` and `@description`. These variables are assigned values in the `initialize` method when a new Thing object is created.

Instance variables generally cannot (and should not) be directly accessed from the world outside the class itself due the principle of encapsulation as explained in the last lesson. In order to obtain the value of each variable we need a *get* accessor method such as `get_name`; in order to assign a new value we need a *set* accessor method such as `set_name`.

SUPERCLASSES AND SUBCLASSES

Now look at the Treasure class. Notice how this is declared:

```
class Treasure < Thing
```

The angle bracket, `<` ,indicates that Treasure is a 'subclass', or descendant, of Thing and therefore it inherits the data (variables) and behaviour (methods) from the Thing class. Since the methods `get_name`, `set_name`, `get_description` and `set_description` already exist in the ancestor class (Thing) these don't need to be re-coded in the descendant class (Treasure).

The Treasure class has one additional piece of data, its value (`@value`) and I have written *get* and *set* accessors for this. When a new Treasure object is created, its `initialize` method is automatically called. A Treasure has three variables to initialize (`@name`, `@description` and `@value`), so its `initialize` method takes three arguments. The first two arguments are passed, using the `super` keyword, to the `initialize` method of the superclass (Thing) so that the Thing class's `initialize` method can deal with them:

```
super( aName, aDescription )
```

When used inside a method, the `super` keyword calls a method with the same name as the current method in the ancestor or 'super' class. If the `super` keyword is used on its own, without any arguments being specified, all the arguments sent to the current method are passed to the ancestor method. If, as in the present case, a specific list of arguments (here `aName` and `aDescription`) is supplied then only these are passed to the method of the ancestor class.

PASSING ARGUMENTS TO THE SUPERCLASS

Brackets matter when calling the superclass! If the argument list is empty and no brackets are used, *all* arguments are passed to the superclass. But if the argument list is empty and brackets are used, *no* arguments are passed to the superclass:

super_args.rb

```
# This passes a, b, c to the superclass
def initialize( a, b, c, d, e, f )
  super( a, b, c )
end

# This passes a, b, c to the superclass
def initialize( a, b, c )
  super
end

# This passes no arguments to the superclass
def initialize( a, b, c)
  super()
end
```

To gain a better understanding of the use of super see the **Digging Deeper** section at the end of this chapter

ACCESSOR METHODS

While the classes in this would-be adventure game work well enough, they are still fairly verbose due to all those *get* and *set* accessors. Let's see what we can do to remedy this.

Instead of accessing the value of the @description instance variable with two different methods, get_description and set_description, like this…

```
puts( t1.get_description )
t1.set_description( "Some description" )
```

…it would be so much nicer to retrieve and assign values just as you would retrieve and assign values to and from a simple variable, like this:

```
puts( t1.description )
t1.description = “Some description”
```

In order to be able to do this, we need to modify the Treasure class definition. One way of accomplishing this would be to rewrite the accessor methods for @description as follows:

```
def description
  return @description
end

def description=( aDescription )
  @description = aDescription
end
```

accessors1.rb

I have added accessors similar to the above in the **accessors1.rb** program. Here, the *get* accessor is called description and the *set* accessor is called description= (that is, it appends an equals sign (=) to the method name used by the corresponding *get* accessor). It is now possible to assign a new string like this:

```
t.description = "a bit faded and worn around the edges"
```

And you can retrieve the value like this:

```
puts( t.description )
```

‘SET’ ACCESSORS

When you write a set accessor in this way, you must append the = character to the method name, not merely place it somewhere between the method name and the arguments.

So this is correct:

```
def name=( aName )
```

But this is an error:

```
def name   =  ( aName )
```

ATTRIBUTE READERS AND WRITERS

In fact, there is a simpler and shorter way of achieving the same result. All you have to do is use two special methods, attr_reader and attr_writer, followed by a symbol like this:

```
attr_reader :description
attr_writer :description
```

You should add this code inside your class definition like this:

```
class Thing
  attr_reader :description
  attr_writer :description
     # maybe some more methods here…
end
```

Calling attr_reader with a symbol has the effect of creating a *get* accessor (here named description) for an instance variable (@description) with a name matching the symbol (:description).

Calling attr_writer similarly creates a *set* accessor for an instance variable. Instance variables are considered to be the 'attributes' of an object, which is why the attr_reader and attr_writer methods are so named.

Symbols

In Ruby, a symbol is a name preceded by a colon (for example, :description). The Symbol class is defined in the Ruby class library to represent names inside the Ruby interpreter. When you pass one or more symbols as arguments to attr_reader (which is a method of the Module class), Ruby creates an instance variable and a *get* accessor method. This accessor method returns the value of the corresponding variable; both the instance variable and the accessor method will take the name that was specified by the symbol. So, attr_reader(:description) creates an instance variable with the name, @description, and an accessor method named description().

accessors2.rb

The **accessors2.rb** program contains some working examples of attribute readers and writers in action. The Thing class explicitly defines a *get* method accessor for the @name attribute. The advantage of writing a complete method like this is that it gives you the opportunity to do some extra processing rather than simply reading and writing an attribute value. Here the *get* accessor uses the String.capitalize method to return the string value of @name with its initial letter in uppercase:

```
def name
  return @name.capitalize
end
```

When assigning a value to the @name attribute, I don't need to do any special processing so I have given it an attribute writer:

```
attr_writer :name
```

The @description attribute needs no special processing so I use attr_reader and attr_writer to get and set the value of the @description variable:

```
attr_reader :description
attr_writer :description
```

Attributes or Properties?

Don't be confused by the terminology. In Ruby, an 'attribute' is the equivalent of what many programming languages call a 'property'.

When you want both to read and to write a variable, the attr_accessor method provides a shorter alternative to using both attr_reader and attr_writer. I have made use of this to access the value attribute in the Treasure class:

```
attr_accessor :value
```

This is equivalent to:

```
attr_reader :value
attr_writer :value
```

Earlier I said that calling attr_reader with a symbol actually creates a variable with the same name as the symbol. The attr_accessor method also does this.

In the code for the Thing class, this behaviour is not obvious since the class has an initialize method which explicitly creates the variables. The Treasure class, however, makes no reference to the @value variable in its initialize method. The only indication that @value exists at all is this accessor definition:

```
attr_accessor :value
```

My code down at the bottom of this source file sets the value of each Treasure object as a separate operation, following the creation of the object itself:

```
t1.value = 800
```

Even though it has never been formally declared, the @value variable really does exist, and we are able to retrieve its numerical value using the *get* accessor:

```
t1.value
```

To be absolutely certain that the attribute accessor really has created @value, you can always look inside the object using the inspect method. I have done so in the final two code lines in this program:

```
puts "This is treasure1: #{t1.inspect}"
puts "This is treasure2: #{t2.inspect}"
```

accessors3.rb

Attribute accessors can initialize more than one attribute at a time if you send them a list of symbols in the form of arguments separated by commas, like this:

```
attr_reader :name, :description
attr_writer(:name, :description)
attr_accessor(:value, :id, :owner)
```

As always, in Ruby, brackets around the arguments are optional but, in my view (for reasons of clarity), are to be preferred.

2adventure.rb

Now let's see how to put attribute readers and writers to use in my adventure game. Load up the **2adventure.rb** program. You will see that I have created two readable attributes in the Thing class: name and description. I have also made description writeable; however, as I don't plan to change the names of any Thing objects, the name attribute is not writeable:

```
attr_reader( :name, :description )
attr_writer( :description )
```

I have created a method called to_s which returns a string describing the Treasure object. Recall that all Ruby classes have a to_s method as standard. The Thing.to_s method overrides (and so replaces) the default one. You can override existing methods when you want to implement new behaviour appropriate to the specific class type.

CALLING METHODS OF A SUPERCLASS

I have decided that my game will have two classes descending from Thing. The Treasure class adds a value attribute which can be both read and written. Note that its initialize method calls its superclass in order to initialize the name and description attributes before initializing the new @value variable:

```
super( aName, aDescription )
@value = aValue
```

Here, if I had omitted the call to the superclass, the name and description attributes would never be initialized. This is because Treasure.initialize overrides Thing.initialize; so when a Treasure object is created, the code in Thing.initialize will not automatically be executed.

On the other hand, the Room class, which also descends from Thing, currently has no initialize method; so when a new Room object is created Ruby goes scrambling back up the class hierarchy in search of one. The first initialize method it finds is in Thing; so a Room object's name and description attributes are initialised there.

CLASS VARIABLES

There are a few other interesting things going on in this program. Right at the top of the Thing class you will see this:

```
@@num_things = 0
```

The two @ characters at the start of this variable name, @@num_things, define this to be a 'class variable'. The variables we've used inside classes up to now have been instance variables, preceded by a single @, like @name. Whereas each new object (or 'instance') of a class assigns its own values to its own instance variables, all objects derived from a specific class share the same class variables. I have assigned 0 to the @@num_things variable to ensure that it has a meaningful value at the outset.

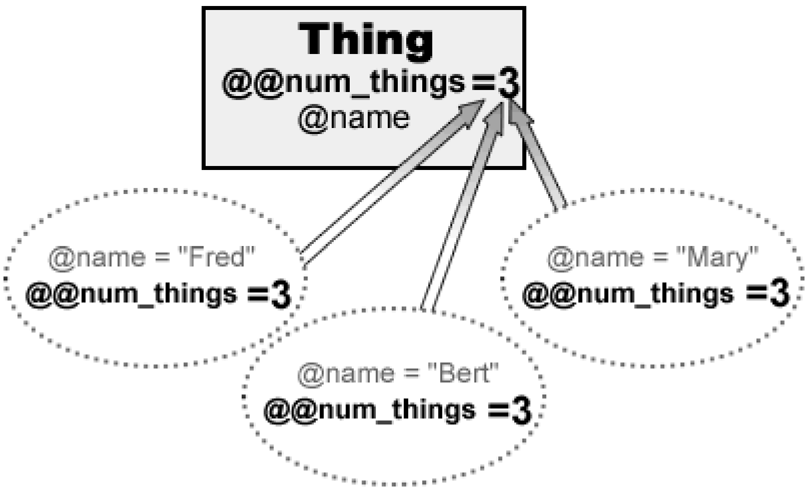
Here, the @@num_things class variable is used to keep a running total of the number of Thing objects in the game. It does this simply by incrementing the class variable (by adding 1 to it: += 1) in its initialize method every time a new object is created:

```
@@num_things +=1
```

If you look lower down in my code, you will see that I have created a Map class to contain an array of rooms. This includes a version of the to_s method which prints information on each room in the array. Don't worry about the implementation of the Map class; we'll be looking at arrays and their methods in a later chapter.

Scroll to the code down at the bottom of the file and run the program in order to see how I have created and initialised all the objects and used the class variable, @@num_things, to keep a tally of all the Thing objects that have been created.

Class Variables and Instance Variables



This diagram shows a Thing class (the rectangle) which contains a class variable, `@@num_things` and an instance variable, `@name`. The three oval shapes represent 'Thing objects' – that is, 'instances' of the Thing class. When one of these objects assigns a value to its instance variable, `@name`, that value only affects the `@name` variable in the object itself – so here, each object has a different value for `@name`. But when an object assigns a value to the class variable, `@@num_things`, that value 'lives inside' the Thing class and is 'shared' by all instances of that class. Here `@@num_things` equals 3 and that is true for all the Thing objects.

Digging Deeper

SUPERCLASSES

super.rb

To understand how the super keyword works, take a look at my sample program, **super.rb**. This contains five related classes: the Thing class is the ancestor of all the others; from Thing descends Thing2; from Thing2 descends Thing3, then Thing4 and Thing5.

Let's take a closer look at the first three classes in this hierarchy: the Thing class has two instance variables, `@name` and `@description`; Thing2 also defines `@fulldescription` (a string which contains `@name` and `@description`); Thing3 adds on yet another variable, `@value`.

These three classes each contain an `initialize` method which sets the values of the variables when a new object is created; they also each have a method named, rather inventively, `aMethod`, which changes the value of one or more variables. The descendant classes, Thing2 and Thing3, both use the `super` keyword in their methods.

Run **super.rb** in a command window. To test out the various bits of code, enter a number, 1 to 5, when prompted or 'q' to quit.

Right down at the bottom of this code unit, I've written a 'main' loop which executes when you run the program. Don't worry about the syntax of this; we'll be looking at loops in a future lesson. I've added this loop so that you can easily run the different bits of code contained in the methods, `test1` to `test5`. When you run this program for the first time, type the number **1** at the prompt and press the Enter key. This will run the `test1` method containing these two lines of code:

```
t = Thing.new( "A Thing", "a lovely thing full of thinginess" )
t.aMethod( "A New Thing" )
```

The first line here creates and initializes a Thing object and the second line calls its `aMethod` method. As the Thing class doesn't descend from anything special (in fact, as with all Ruby classes, it descends from the Object class which is the ultimate ancestor of all other classes) nothing very new or interesting happens here. The output uses the inspect method to display the internal structure of the object when the Thing.initialize and Thing.aMethod methods are called. The inspect method can be used with all objects and is an invaluable debugging aid. Here, it shows us a hexadecimal number which identifies this specific object followed by the string values of the @name and @description variables.

Now, at the prompt, enter **2** to run test2 containing this code to create a Thing2 object, t2, and call t2.aMethod:

```
t2 = Thing2.new( "A Thing2", "a Thing2 thing of great beauty" )
t2.aMethod( "A New Thing2", "a new Thing2 description" )
```

Look carefully at the output. You will see that even though t2 is a Thing2 object, it is the Thing class's initialize method that is called first. To understand why this is so, look at the code of the Thing2 class's initialize method:

```
def initialize( aName, aDescription )
  super
  @fulldescription = "This is #{@name}, which is #{@description}"
  puts("Thing2.initialize: #{self.inspect}\n\n")
end
```

This uses the super keyword to call the initialize method of Thing2's ancestor or 'superclass'. The superclass of Thing2 is Thing as you can see from its declaration:

```
class Thing2 < Thing
```

In Ruby, when the super keyword is used on its own (that is, without any arguments), it passes all the arguments from the current method (here Thing2.initialize) to a method with the same name in its superclass (here Thing.initialize). Alternatively, you can explicitly specify a list of arguments following super. So, in the present case, the following code would have the same effect:

```
super( aName, aDescription )
```

While it is permissible to use the super keyword all on its own, in my view it is often preferable, for the sake of clarity, explicitly to specify the list of arguments to be passed to the superclass. At any rate, if you want to pass only a limited number of the arguments sent to the current method, an explicit argument list is necessary. Thing2's aMethod, for example, only passes the aName argument to the initialize method of its superclass, Thing1:

```
super( aNewName )
```

This explains why the @description variable is not changed when the method, Thing2.aMethod, is called.

Now if you look at Thing3 you will see that this adds on one more variable, @value. In its implementation of initialize it passes the two arguments, aName and aDescription to its superclass, Thing2. In its turn, as we've already seen, Thing2's initialize method passes these same arguments to the initialize method of its superclass, Thing.

With the program running, enter **3** at the prompt to view the output. This is the code which executes this time:

```
t3 = Thing3.new("A Thing 3", "a Thing3 full of Thing and
     Thing2iness",500)
t3.aMethod( "A New Thing3", "and a new Thing3 description",1000)
```

Note how the flow of execution goes right up the hierarchy so that code in the initialize and aMethod methods of Thing execute before code in the matching methods of Thing2 and Thing3.

It is not obligatory to a override superclass's methods as I have done in the examples so far. This is only required when you want to add some new behaviour. Thing4 omits the initialize method but implements the aMethod method.

Enter **4** at the prompt to execute the following code:

```
t4 = Thing4.new( "A Thing4", "the nicest Thing4 you will ever see", 10 )
t4.aMethod
```

When you run it, notice that the first available initialize method is called when a Thing4 object is created. This happens to be Thing3.initialize which, once again, also calls the initialize methods of its ancestor classes, Thing2 and Thing. However, the aMethod method implemented by Thing4 has no call to its superclasses, so this executes right away and the code in any other aMethod methods in the ancestor classes is ignored.

Finally, Thing5 inherits from Thing4 and doesn't introduce any new data or methods. Enter 5 at the prompt to execute the following:

```
t5 = Thing5.new( "A Thing5", "a very simple Thing5", 40 )
t5.aMethod
```

This time you will see that that the call to new causes Ruby to backtrack through the class hierarchy until it finds the first initialize method. This happens to belong to Thing3 (which also calls the initialize methods of Thing2 and Thing). The first implementation of aMethod, however, occurs in Thing4 and there are no calls to super so that's where the trail ends.

superclasses.rb

Ultimately all Ruby classes descend from the Object class.

The Object class itself has no superclass and any attempt to locate its superclass will return *nil*.

```
begin
  x = x.superclass
  puts(x)
end until x == nil
```

CONSTANTS INSIDE CLASSES

There may be times when you need to access constants (identifiers beginning with a capital letter) declared inside a class. Let's assume you have this class:

classconsts.rb

```
class X
  A = 10

  class Y
  end
end
```

In order to access the constant A, you would need to use the special scope resolution operator :: like this:

```
X::A
```

Class names are constants, so this same operator gives you access to classes inside other classes. This makes it possible to create objects from 'nested' classes such as class Y inside class X:

```
ob = X::Y.new
```

PARTIAL CLASSES

In Ruby it is not obligatory to define a class all in one place. If you wish, you can define a single class in separate parts of your program. When a class descends from a specific superclass, each subsequent partial class definition may optionally repeat the superclass in its definition using the < operator.

Here I create two classes, A and B which descends from A:

partial_classes

```
class A
  def a
    puts( "a" )
  end
end

class B < A
  def ba1
    puts( "ba1" )
  end
end

class A
  def b
    puts( "b" )
  end
end

class B < A
  def ba2
    puts( "ba2" )
  end
end
```

Now, if I create a B object, all the methods of both A and B are available to it:

```
ob = B.new
ob.a
ob.b
ob.ba1
ob.ba2
```

You can also use partial class definitions to add features onto Ruby's standard classes such as Array:

```
class Array
  def gribbit
    puts( "gribbit" )
  end
end
```

This adds the `gribbit` method to the Array class so that the following code can now be executed:

```
[1,2,3].gribbit
```

CHAPTER THREE

Strings and Ranges

I've made great use of strings in my programs so far. In fact, a string featured in the very first program in the book. Here it is again:

```
puts 'hello world'
```

While that first program used a string enclosed within single quotes, my second program rang the changes by using a string in double-quotes:

```
print('Enter your name: ' )
name = gets()
puts( "Hello #{name}" )
```

1strings.rb

Double-quoted strings do more work than single-quoted strings. In particular, they have the ability to evaluate bits of themselves as though they were programming code. To have something evaluated, you need to place it between a pair of curly braces preceded by a # character.

In the example above, `#{name}` in a double-quoted string tells Ruby to get the value of the `name` variable and insert that value into the string itself. So, if `name` equals "Fred", the string "Hello Fred" will be displayed. The **1strings.rb** sample program provides a few more examples of 'embedded evaluation' in double-quoted strings.

A double-quoted string is able not only to evaluate attributes or variables such as `ob.name` but also expressions such as `2*3` and bits of code such as the method-call ob.ten (where ten is a method name) and 'escape characters' such as "\n" and "\t" representing a newline and a tab.

A single-quoted string does no such evaluation. A single-quoted string can, however, use a backslash to indicate that the next character should be used literally. This is useful when a single-quoted string contains a single-quote character, like this:

```
'It\'s my party'
```

Assuming that the method named ten returns the value 10, you might write the following code:

```
puts( "Here's a tab\ta new line\na calculation #{2*3} and a method-call
#{ob.ten}" )
```

As this is a double-quoted string, the embedded elements are evaluated and the following is displayed:

```
Here's a tab        a new line
a calculation 6 and a method-call 10
```

Now let's see what happens when a single-quoted string is used:

```
puts( 'Here\'s a tab\ta new line\na calculation #{2*3} and a method-call
#{ob.ten}' )
```

This time, no embedded evaluation is done and so, this is what is displayed:

```
Here's a tab\ta new line\na calculation #{2*3} and a method-call
#{ob.ten}
```

USER-DEFINED STRING DELIMITERS

If, for some reason, single and double-quotes aren't convenient – if, for example, your strings contain lots of quote characters and you don't want to have to keep putting backslashes in front of them – you can also delimit strings in many other ways.

2strings.rb

The standard alternative delimiters for double quoted strings are %Q and / or %/ and / while for single-quoted strings they are %q and /. Thus…

```
%Q/This is the same as a double-quoted string./
%/This is also the same as a double-quoted string./
%q/And this is the same as a single-quoted string/
```

You can even define your own string delimiters. These must be non-alphanumeric characters and they may include non-printing characters such as newlines and various characters which normally have a special meaning in Ruby such as the 'pound' or 'hash' character (#). Your chosen character should be placed after %q or %Q and you should be sure to terminate the string with the same character. If your delimiter is an opening bracket, the corresponding closing bracket should be used at the end of the string, like this:

```
%Q[This is a string]
```

3strings.rb

You will find examples of a broad range of user-selected string delimiters in the sample program, **3strings.rb**. Needless to say, while there may be times when it is useful to delimit a string by some esoteric character such as a newline or an asterisk, in many cases the disadvantages (not least the mental anguish and confusion) resulting from such arcane practices may significantly outweigh the advantages.

BACKQUOTES

One other type of string deserves a special mention: a string enclosed by back-quotes – that is, the inward-pointing quote character which is usually tucked away up towards the top left-hand corner of the keyboard: `

Ruby considers anything enclosed by back-quotes to be a command which can be passed for execution by the operating system using a method such as `print` or `puts`. By now, you will probably already have guessed that Ruby provides more than one way of doing this. It turns out `%x/some command/` has the same effect as `somecommand` and so does %x{some command}. On the Windows operating system, for example, each of the three lines shown below would pass the command dir to the operating system, causing a directory listing to be displayed:

4backquotes.rb

```
puts(`dir`)
puts(%x/dir/)
puts(%x{dir})
```

You can also embed commands inside double-quoted strings like this:

```
print( "Goodbye #{%x{calc}}" )
```

Be careful if you do this. The command itself is evaluated first. Your Ruby program then waits until the process which starts has terminated. In the present case, the calculator will pop up. You are now free to do some calculations, if you wish. Only when you close the calculator will the string "Goodbye" be displayed.

STRING HANDLING

Before leaving the subject of strings, we'll take a quick look at a few common string operations.

CONCATENATION

string_concat.rb

You can concatenate strings using << or + or just by placing a space between them. Here are three examples of string concatenation; in each case, s is assigned the string "Hello World":

```
s = "Hello " << "world"
s = "Hello " + "world"
s = "Hello "  "world"
```

Note, however, that when you use the << method, you can append Fixnum integers (in the range 0 to 255) without having to convert them to strings first; using the + method or a space, Fixnums must be converted using the to_s method.

What About Commas?

You may sometimes see Ruby code in which commas are used to separate strings and other data types. In some circumstances, these commas appear to have the effect of concatenating strings. For example, the following code might, at first sight, seem to create and display a string from three substrings plus an integer:

```
s4 = "This " , "is" , " not a string!", 10
print("print (s4):" , s4, "\n")
```

In fact, a list separate by commas creates an array – an ordered list of the original strings. The **string_concat.rb** program contains examples which prove this to be the case.

Note that when you pass an array to a method such as puts, each element in that array will be treated separately. You could pass the array, x, above, to puts like this:

```
puts( x )
```

In which case, the output would be:

```
This
is
 not a string!
10
```

We'll look at arrays in more depth in the next chapter.

STRING ASSIGNMENT

The Ruby String class provides a number of useful string handling methods. Most of these methods create new string objects. So, for example, in the following code, the s on the left-hand side of the assignment on the second line is not the same object as the s on the right-hand side:

```
s = "hello world"
s = s + "!"
```

string_assign.rb

A few string methods actually alter the string itself without creating a new object. These methods generally end with an exclamation mark (e.g. the capitalize! method).

If in doubt, you can check an object's identity using the object_id method. I've provided a few examples of operations which do and do not create new strings in the **string_assign.rb** program. Run this and check the object_id of s after each string operation is performed.

INDEXING INTO A STRING

You can treat a string as an array of characters and index into that array to find a character at a specific index using square brackets. Strings and arrays in Ruby are indexed from 0 (the first character). So, for instance, to replace the character 'e' with 'a' in the string, s, which currently contains 'Hello world', you would assign a new character to index 1:

```
s[1] = 'a'
```

However, if you index into a string in order to find a character at a specific location, Ruby doesn't return the character itself; it returns its ASCII value:

```
s = "Hello world"
puts( s[1] )                # prints out 101 – the ASCII value of 'e'
```

In order to obtain the actual character, you can do this:

```
s = "Hello world"
puts( s[1,1] )        # prints out 'e'
```

This tells Ruby to index into the string at position 1 and return one character. If you want to return three characters starting at position 1, you would enter this:

```
puts( s[1,3] )        # prints 'ell'
```

This tells Ruby to start at position 1 and return the next 3 characters. Alternatively, you could use the two-dot 'range' notation:

```
puts( s[1..3] )       # also prints 'ell'
```

For more on Ranges, see **Digging Deeper** at the end of this chapter.

Strings can also be indexed using minus values, in which case -1 is the index of the last character and, once again, you can specify the number of characters to be returned:

```
puts( s[-1,1] )       # prints 'd'
puts( s[-5,1] )       # prints 'w'
puts( s[-5,5] )       # prints 'world'
```

string_index.rb

When specifying ranges using a minus index, you must use minus values for both the start and end indices:

```
puts( s[-5..5] )      # this prints an empty string!
puts( s[-5..-1] )     # prints 'world'
```

string_methods.rb

Finally, you may want to experiment with a few of the standard methods available for manipulating strings. These include methods to change the case of a string, reverse it, insert substrings, remove repeating characters and so on. I've provided a few examples in **string_methods.rb**.

REMOVING NEWLINE CHARACTERS – CHOP AND CHOMP

A couple of handy string processing methods deserve special mention. The chop and chomp methods can be used to remove characters from the end of a string. The chop method returns a string with the last character removed or with the carriage return and newline characters removed ("\r\n") if these are found at the end of the string. The chomp method returns a string with the terminating carriage return or newline character removed (or both the carriage return *and* the newline character if both are found).

These methods are useful when you need to removing line feeds entered by the user or read from a file. For instance, when you use gets to read in a line of text, it returns the line including the terminating 'record separator' which, by default, is the newline character.

The Record Separator - $/

Ruby pre-defines a variable, $/, as a 'record separator'. This variable is used by methods such as gets and chomp. The gets method reads in a string up to and including the record separator. The chomp method returns a string with the record separator removed from the end (if present) otherwise it returns the original string unmodified. You can redefine the record separator if you wish, like this:

```
$/="*"              # the "*" character is now the record separator
```

When you redefine the record separator, this new character (or string) will now be used by methods such as gets and chomp. For example:

```
$/= "world"
s = gets()          # user enters "Had we but world enough and
time…"
puts( s )           # displays "Had we but world"
```

You can remove the newline character using either chop or chomp. In most cases, chomp is preferable as it won't remove the final character unless it is the record separator (a newline) whereas chop will remove the last character no matter what it is. Here are some examples:

chop_chomp.rb

```
# Note: s1 includes a carriage return and linefeed
s1 = "Hello world
"
s2 = "Hello world"
s1.chop          # returns "Hello world"
s1.chomp         # returns "Hello world"
s2.chop          # returns "Hello worl" – note the missing 'd'!
s2.chomp         # returns "Hello world"
```

The `chomp` method lets you specify a character or string to use as the separator:

```
s2.chomp('rld')     # returns "Hello wo"
```

FORMAT STRINGS

Ruby provides the `printf` method to print 'format strings' containing specifiers starting with a percent sign, `%`. The format string may be followed by one or more data items separated by commas; the list of data items should match the number and type of the format specifiers. The actual data items replace the matching specifiers in the string and they are formatted accordingly. These are some common formatting specifiers:

```
%d – decimal number
%f – floating point number
%o – octal number
%p – inspect object
%s – string
%x – hexadecimal number
```

You can control floating point precision by putting a point-number before the floating point formatting specifier, `%f`. For example, this would display the floating point value to two digits:

```
printf( "%0.02f", 10.12945 )     # displays 10.13
```

Digging Deeper

RANGES

In Ruby, a Range is a class which represents a set of values defined by a starting and an ending value. Typically a range is defined using integers but it may also be defined using other ordered values such as floating point numbers or characters. Values can be negative, though you should be careful that your starting value is lower than your ending value!

ranges.rb

Here are a few examples:

```
a = (1..10)
b = (-10..-1)
c = (-10..10)
d = ('a'..'z')
```

You can also specify ranges using three dots instead of two: this create a range which omits the final value:

```
d = ('a'..'z')                # this two-dot range = 'a'..'z'
e = ('a'...'z')               # this three-dot range = 'a'..'y'
```

You can create an array of the values defined by a range using the `to_a` method, like this:

```
(1..10).to_a
```

Note that `to_a` is not defined for floating point numbers for the simple reason that the number of possible values between two floating point numbers is not finite.

str_range.rb

You can even create ranges of strings – though you would need to take great care in so doing as you might end up with more than you bargain for. For example, see if you can figure out which values are specified by this range:

```
str_range = ('abc'..'def')
```

At first sight, the range from 'abc' to 'def' might not look much. In fact, this defines a range of no less than 2,110 values! They are ordered like this: 'abc', 'abd', 'abe' and so on until the end of the 'a's; then we start on the 'b's: 'baa', 'bab', 'bac' and so on. Suffice to say that ranges of this sort are probably rather a rare requirement and are best used with extreme caution or not at all.

ITERATING WITH A RANGE

You may use a range to iterate from a start value to an end value. For example, here is one way of printing all the numbers from 1 to 10:

for_to.rb

```
for i in (1..10) do
  puts( i )
end
```

HEREDOCS

While you can write long strings spanning multiple lines between single or double quotes, many Ruby programmers prefer to use an alternative type of string called a 'heredoc'. A heredoc is a block of text that starts by specifying an end marker, which is simply an identifier of your choice. Here, I specify EODOC as the end marker:

heredoc.rb

```
hdoc1 = <<EODOC
```

This tells Ruby that everything following the line above is a single string which terminates when the end marker is located. The string is assigned to the variable, hdoc1. Here is an example of a complete heredoc assignment:

```
hdoc1 = <<EODOC
I wandered lonely as a #{"cloud".upcase},
That floats on high o'er vale and hill...
EODOC
```

By default, heredocs are treated as double-quoted strings so expressions such as #{"cloud".upcase} will be evaluated. If you want a heredoc to be treated as single-quoted string, specify its end marker between single-quotes:

```
hdoc2 = <<'EODOC'
I wandered lonely as a #{"cloud".upcase},
That floats on high o'er vale and hill...
EODOC
```

The end-marker of a heredoc must, by default, be placed flush with the left margin. If you want to indent if you should use <<- rather than << when assigning the end marker:

```
hdoc3 = <<-EODOC
I wandered lonely as a #{"cloud".upcase},
That floats on high o'er vale and hill...
    EODOC
```

It is up to you to pick an appropriate end marker. It is even legitimate (though, perhaps, not particularly sensible!) to use a reserved word:

```
hdoc4 = <<def
I wandered lonely as a #{"cloud".upcase},
That floats on high o'er vale and hill...
def
```

A variable to which a heredoc is assigned can be used just like any other string variable:

```
puts( hdoc1 )
```

STRING LITERALS

As explained earlier in this chapter, you can optionally delimit strings by %q/ and / for single-quoted strings and either %Q/ and / or %/ and / for double-quoted strings.

Ruby provides similar means of delimiting back-quoted strings, regular expressions, symbols and arrays of either single-quoted or double-quoted strings. The ability to define arrays of strings in this way is particularly useful since it avoids the necessity of entering string delimiters for each item. Here is a reference to these string literal delimiters:

```
%q/     /
%Q/     /
%/      /
%w/     /
%W/     /
%r|     |
%s/     /
%x/     /
```

Note that you may choose which delimiters to use. I have used / except with the regular expression where I have used | (since / is the 'normal' regular expression delimiter) but I could equally have used square brackets, asterisks, ampersands or other symbols (e.g. %W*dog cat #{1+2}* or %s&dog&). Here is an example of these literals in use:

literals.rb

```
p %q/dog cat #{1+2}/          #=> "dog cat \#{1+2}"
p %Q/dog cat #{1+2}/          #=> "dog cat 3"
p %/dog cat #{1+2}/           #=> "dog cat 3"
p %w/dog cat #{1+2}/          #=> ["dog", "cat", "\#{1+2}"]
p %W/dog cat #{1+2}/          #=> ["dog", "cat", "3"]
p %r|^[a-z]*$|                #=> /^[a-z]*$/
p %s/dog/                     #=> :dog
p %x/vol/                     #=> " Volume in drive C is OS [etc...]"
```

CHAPTER FOUR

Arrays and Hashes

Up to now, we've generally been using objects one at a time. In this chapter we'll find out how to create a list of objects. We'll start by looking at the most common type of list structure – an array.

ARRAYS

array0.rb

What is an Array?

An Array is a sequential collection of items in which each item can be indexed. In Ruby, (unlike many other languages) a single Array can contain items of mixed data types such as strings, integers and floats or even a method-call which returns some value:

```
a1 = [1,'two', 3.0, array_length( a0 ) ]
```

The first item in an array has the index 0, which means that the final item has an index equal to the total number of items in the array minus 1. Given the array, `a1`, shown above, this is how to obtain the values of the first and last items:

```
a1[0]          # returns 1st item (at index 0)
a1[3]          # returns 4th item (at index 3)
```

We've already used arrays a few times – for example, in **2adventure.rb** in chapter 2 we used an array to store a map of Rooms:

```
mymap = Map.new([room1,room2,room3])
```

CREATING ARRAYS

In common with many other programming languages, Ruby uses square brackets to delimit an array. You can easily create an array, fill it with some comma-delimited values and assign it to a variable:

```
arr = ['one','two','three','four']
```

array1.rb

As with most other things in Ruby, arrays are objects. They are defined, as you might guess, by the Array class and, just like strings, they are indexed from 0. You can reference an item in an array by placing its index between square brackets. If the index is invalid, nil is returned:

```
arr = ['a', 'b', 'c']
puts(arr[0])        # shows 'a'
puts(arr[1])        # shows 'b'
puts(arr[2])        # shows 'c'
puts(arr[3])        # nil
```

array2.rb

It is permissible to mix data types in an array and even to include expressions which yield some value. Let's assume that you have already created this method:

```
def hello
  return "hello world"
end
```

You can now declare this array:

```
x = [1+2, hello, `dir`]
```

Here, the first element is the integer, 3 and the second is the string "hello world" (returned by the method `hello`). If you run this on Windows, the third array element will be a string containing a directory listing. This is due to the fact that `` `dir` `` is a back-quoted string which is executed by the operating system (*see Chapter 3*). The final 'slot' in the array is, therefore, filled with the value returned by the **dir** command which happens to be a string of file names. If you are running on a different operating system, you may need to substitute an appropriate command at this point.

dir_array.rb

Creating an Array of File Names

A number of Ruby classes have methods which return arrays of values. For example, the Dir class, which is used to perform operations on disk directories, has the `entries` method. Pass a directory name to the method and it returns a list of files in an array:

```
Dir.entries( 'C:\\' )    # returns an array of files in C:\
```

If you want to create an array of single-quoted strings but can't be bothered typing all the quotation marks, a shortcut is to put unquoted text separated by spaces between round brackets preceded by `%w` like this (or use a capital `%W` for double-quoted strings, as explained in Chapter 3):

array2.rb

```
y = %w( this is an array of strings )
```

You can also create arrays using the usual object construction method, `new`. Optionally, you can pass an integer to `new` to create an empty array of a specific size (with each element set to `nil`), or you can pass two arguments – the first to set the size of the array and the second to specify the element to place at each index of the array, like this:

```
a = Array.new                    # an empty array
a = Array.new(2)                 # [nil,nil]
a = Array.new(2,"hello world")   # ["hello world","hello world"]
```

MULTI-DIMENSIONAL ARRAYS

To create a multi-dimensional array, you can create one array and then add other arrays to each of its 'slots'. For example, this creates an array containing two elements, each of which is itself an array of two elements:

```
a = Array.new(2)
a[0]= Array.new(2,'hello')
a[1]= Array.new(2,'world')
```

> You can also create an Array object by passing an array as an argument to the new method. Be careful, though. It is a quirk of Ruby that, while it is legitimate to pass an array argument either with or without enclosing round brackets, Ruby considers it a syntax error if you fail to leave a space between the new method and the opening square bracket – another good reason for making a firm habit of using brackets when passing arguments!

It is also possible to nest arrays inside one another using square brackets. This creates an array of four arrays, each of which contains four integers:

```
a = [       [1,2,3,4],
            [5,6,7,8],
            [9,10,11,12],
            [13,14,15,16]  ]
```

In the code shown above, I have placed the four 'sub-arrays' on separate lines. This is not obligatory but it does help to clarify the structure of the multi-dimensional array by displaying each sub-array as though it were a row, similar to the rows in a spreadsheet. When talking about arrays within arrays, it is convenient to refer to each nested array as a 'row' of the 'outer' array.

multi_array.rb

For some more examples of using multi-dimensional arrays, load up the **multi_array.rb** program. This starts by creating an array, `multiarr`, containing two other arrays. The first of these arrays is at index 0 of `multiarr` and the second is at index 1:

```
multiarr = [['one','two','three','four'],[1,2,3,4]]
```

ITERATING OVER ARRAYS

You can access the elements of an array by iterating over them using a `for` loop. The loop will iterate over two elements here: namely, the two sub-arrays at index 0 and 1:

```
for i  in multiarr
  puts(i.inspect)
end
```

This displays:

```
["one", "two", "three", "four"]
[1, 2, 3, 4]
```

So, how do you iterate over the items (the strings and integers) in each of the two sub-arrays? If there is a fixed number of items you could specify a different iterator variable for each, in which case each variable will be assigned the value from the matching array index.

Here we have four sub-array slots, so you could use four variables like this:

```
for (a,b,c,d) in multiarr
     print("a=#{a}, b=#{b}, c=#{c}, d=#{d}\n" )
end
```

Iterators and for loops

The code inside a for loop is executed for each element in some expression. The syntax can be summarized like this:

```
for <one or more variables> in <expression> do
  <code to run>
end
```

When more than one variable is supplied, these are passed to the code inside the for..end block just as you would pass arguments to a method. Here, for example, you can think of (a,b,c,d) as four arguments which are initialised, at each turn through the for loop, by the four values from a row of multiarr:

```
for (a,b,c,d) in multiarr
  print("a=#{a}, b=#{b}, c=#{c}, d=#{d}\n" )
end
```

We'll be looking at for loops and other iterators in more depth in the next chapter.

multi_array2.rb

You could also use a for loop to iterate over all the items in each sub-array individually:

```
for s in multiarr[0]
  puts(s)
end
for s in multiarr[1]
  puts(s)
end
```

Both of the above techniques (multiple iterator variables or multiple for loops) have two requirements: a) that you know how many items there are either in the 'rows' or 'column's of the grid of arrays and b) that each sub array contains the same number of items as each other.

For a more flexible way of iterating over multidimensional arrays you could use nested `for` loops. An outer loop iterates over each row (subarray) and an inner loop iterates over each item in the current row. This technique works even when subarrays have varying numbers of items:

```
for row  in multiarr
  for item in row
    puts(item)
  end
end
```

INDEXING INTO ARRAYS

As with strings (*see Chapter Three*) , you can index from the end of an array using minus figures, where -1 is the index of the last element; and you can also use ranges:

array_index.rb

```
arr = ['h','e','l','l','o',' ','w','o','r','l','d']

print( arr[0,5] )           #=> 'hello'
print( arr[-5,5 ] )         #=> 'world'
print( arr[0..4] )          #=> 'hello'
print( arr[-5..-1] )        #=> 'world'
```

Notice that, as with strings, when provide two integers in order to return a number of contiguous items from an array, the first integer is the start index while the second is a *count* of the number of items (*not* an index):

```
arr[0,5]                    # returns 5 chars - ["h", "e", "l", "l", "o"]
```

array_assign.rb

You can also make assignments by indexing into an array. Here, for example, I first create an empty array then put items into indexes 0, 1 and 3. The 'empty' slot at number 2 will be filled with a `nil` value:

```
arr = []

arr[0] = [0]
arr[1] = ["one"]
arr[3] = ["a", "b", "c"]

# arr now contains:
# [[0], ["one"], nil, ["a", "b", "c"]]
```

Once again, you can use start-end indexes, ranges and negative index values:

```
arr2 = ['h','e','l','l','o',' ','w','o','r','l','d']

arr2[0] = 'H'
arr2[2,2] = 'L', 'L'
arr2[4..6] = 'O','-','W'
arr2[-4,4] = 'a','l','d','o'

# arr2 now contains:
# ["H", "e", "L", "L", "O", "-", "W", "a", "l", "d", "o"]
```

COPYING ARRAYS

array_copy.rb

Note that when you use the assignment operator, =, to assign one array variable to another variable, you are actually assigning a reference to the array itself – you are not making a copy. You can use the clone method to make a new copy of the array:

```
arr1=['h','e','l','l','o',' ','w','o','r','l','d']
arr2=arr1
     # arr2 is now the same as arr1. Change arr1 and arr2 changes too!
arr3=arr1.clone
     # arr3 is a copy of arr1. Change arr1 and arr2 is unaffected
```

TESTING ARRAYS FOR EQUALITY

array_compare.rb

A few words need to be said about the comparison operator `<=>`. This compares two arrays – let's call them `arr1` and `arr2`; it returns -1 if `arr1` is less than `arr2`; it returns 0 if `arr1` and `arr2` are equal; it returns 1 if `arr2` is greater than `arr1`. But how does Ruby determine if one array is 'greater than' or 'less than' another? It turns out that it compares each item in one array with the corresponding item in the other. When two values are not equal, the result of their comparison is returned. In other words if this comparison were made:

```
[0,10,20] <=> [0,20,20]
```

…the value -1 would be returned (the first array is 'less than' the second) since the integer at index 1 is of lower value (10) in the first array than the integer at the same index in the second (20).

If you are comparing arrays of strings, then comparisons are made on ASCII values. If one array is longer than another and the elements in both arrays are all equal, then the longer array is deemed to be 'greater'. However, if two such arrays are compared and one of the elements in the shorter array is greater than the corresponding element in the longer array, then the *shorter* array is deemed to be greater.

SORTING ARRAYS

array_sort.rb

The `sort` method compares adjacent array elements using the comparison operator `<=>`. This operator is defined for many Ruby classes, including Array, String, Float, Date and Fixnum. The sort operator is not, however, defined for *all* classes (that is to say that it is not defined for the Object class from which all other classes are derived). One of the unfortunate consequences of this is that it cannot be used to sort arrays containing `nil` values. It is, however, possible to get around this limitation by defining your own sorting routine. This is done by sending a block to the sort method. We'll look at blocks in detail in Chapter 10. For now, it is enough to know that the block here is a chunk of code which determines the comparison used by the `sort` method.

This is my sort routine:

```
arr.sort{
  |a,b|
    a.to_s <=> b.to_s
}
```

Here arr is an array object and the variables a and b represent two contiguous array elements. I've converted each variable to a string using the to_s method; this converts nil to an empty string which will be sorted 'low'. Note that, while my sorting block defines the sort order of the array items, it does not change the array items themselves. So nil will remain as nil and integers will remain as integers. The string conversion is only used to implement the comparison, not to change the array items.

COMPARING VALUES

The comparison 'operator' <=> (which is, in fact, a method) is defined in the Ruby module named Comparable. For now, you can think of a module as a sort of reusable 'code library'. We'll be looking more closely at modules in Chapter 12.

You can 'include' the Comparable module in your own classes. When this is done, you can override the <=> method to enable you to define exactly how comparisons will be made between specific types of object. For example, you may want to subclass Array so that comparisons are made based purely on the length of two Arrays rather than on the values of each item in the Array (which is the default, as explained earlier). This is how to might do this:

comparisons.rb

```
class MyArray < Array
  include Comparable

  def <=> ( anotherArray )
    self.length <=> anotherArray.length
  end
end
```

Now, you can initialize two MyArray objects like this:

```
myarr1 = MyArray.new([0,1,2,3])
myarr2 = MyArray.new([1,2,3,4])
```

And you can use the <=> method defined in MyArray in order to make comparisons:

```
                            # Two MyArray objects
myarr1 <=> myarr2           # returns 0
```

This returns 0 which indicates that the two arrays are equal (since our <=> method evaluates equality according to length alone). If, on the other hand, we were to initialise two standard Arrays with exactly the same integer values, the Array class's own <=> method would perform the comparsion:

```
                            # Two Array objects
arr1 <=> arr2               # returns -1
```

Here -1 indicates that the first array evaluates to 'less than' the second array since the Array class's <=> method compares the numerical values of each item in arr1 and these are less than the values of the items at the same indexes in arr2.

But what if you want to make 'less than', 'equal to' and 'greater than' comparisons using the traditional programming notation:

```
<                           # less than
==                          # equal to
>                           # greater than
```

In the MyArray class, we can make comparisons of this sort without writing any additional code. This is due to the fact that the Comparable module, which has been included in the MyArray class, automatically supplies these three comparison methods; each method makes its comparison based on the definition of the <=> method. Since our <=> makes its evaluation based on the number of items in an array, the < method evaluates to true when the first array is shorter than the second, == evaluates to true when both arrays are of equal length and > evaluates to true when the second array is longer than the first:

```
p( myarr1 < myarr2 )       #=> false
p( myarr1 == myarr2 )      #=> true
```

The standard Array, class, however, does not include the Comparable module so, if you try to compare two ordinary arrays using <, == or >, Ruby will display an error message telling you that the method is undefined.

It turns out that it is easy to add these three methods to a subclass of Array. All you have to do is include Comparable, like this:

```
class Array2 < Array
  include Comparable
end
```

The Array2 class will now perform its comparisons based on the <=> method of Array – that is, by testing the values of the items stored in the array rather than merely testing the length of the array. Assuming the Array2 objects, arr1 and arr2, to be initialized with the same arrays which we previously used for myarr1 and myarr2, we would now see these results:

```
p( arr1 < arr2 )           #=> true
p( arr1 > arr2 )           #=> false
```

ARRAY METHODS

array_methods.rb

Several of the standard array methods modify the array itself rather than returning a modified copy of the array. These include not only those methods marked with a terminating exclamation such as flatten! and compact! but also the method << which modifies the array to its left by adding to it the array on its right; the clear which removes all the elements from the array and delete and delete_at remove selected elements.

HASHES

While arrays provide a good way of indexing a collection of items by number, there may be times when it would be more convenient to index them in some other way. If, for example, you were creating a collection of recipes, it would be more meaningful to have each recipe indexed by name such as "Rich Chocolate Cake" and "Coq au Vin" rather than by numbers: 23, 87 and so on.

Ruby has a class that lets you do just that. It's called a Hash. This is the equivalent of what some other languages call a Dictionary. Just like a real dictionary, the entries are indexed by some unique key (in a dictionary, this would be a word) which is associated with a value (in a dictionary, this would be the definition of the word).

CREATING HASHES

hash1.rb

Just like an array, you can create a hash by creating a new instance of the Hash class:

```
h1 = Hash.new
h2 = Hash.new("Some kind of ring")
```

Both the examples above create an empty Hash. A Hash object always has a default value – that is, a value that is returned when no specific value is found at a given index. In these examples, `h2` is initialized with the default value, "Some kind of ring"; `h1` is not initialized with a value so its default value will be `nil`.

Having created a Hash object, you can add items to it using an array-like syntax – that is, by placing the index in square brackets and using `=` to assign a value. The obvious difference here being that, with an array, the index (the 'key') must be an integer; with a Hash, it can be any unique data item:

```
h2['treasure1'] = 'Silver ring'
h2['treasure2'] = 'Gold ring'
h2['treasure3'] = 'Ruby ring'
h2['treasure4'] = 'Sapphire ring'
```

Often, the key may be a number or, as in the code above, a string. In principle, however, a key can be any type of object.

Unique Keys?

Take care when assigning keys to Hashes. If you use the same key twice in a Hash, you will end up overwriting the original value. This is just like assigning a value twice to the same index in an array. Consider this example:

```
h2['treasure1'] = 'Silver ring'
h2['treasure2'] = 'Gold ring'
h2['treasure3'] = 'Ruby ring'
h2['treasure1'] = 'Sapphire ring'
```

Here the key 'treasure1' has been used twice. As a consequence, the original value, 'Silver ring' has been replaced by 'Sapphire ring', resulting in this Hash:

```
{"treasure1"=>"Sapphire ring", "treasure2"=>"Gold ring", "treas-
ure3"=>"Ruby ring"}
```

Given some class, X, the following assignment is perfectly legal:

```
x1 = X.new('my Xobject')
h2[x1] = 'Diamond ring'
```

There is a shorthand way of creating Hashes and initializing them with key-value pairs. Just add a key followed by => and its associated value; each key-value pair should be separated by a comma and the whole lot placed inside a pair of curly brackets:

```
h1 = {       'room1'=>'The Treasure Room',
             'room2'=>'The Throne Room',
             'loc1'=>'A Forest Glade',
             'loc2'=>'A Mountain Stream' }
```

INDEXING INTO A HASH

To access a value, place its key between square brackets:

```
puts(h1['room2'])                #=> 'The Throne Room'
```

If you specify a key that does not exist, the default value is returned. Recall that we have not specified a default value for h1 but we have for h2:

```
p(h1['unknown_room'])            #=> nil
p(h2['unknown_treasure'])        #=> 'Some kind of ring'
```

Use the default method to get the default value and the default= method to set it (see Chapter 2 for more information on *get* and *set* 'accessor' methods):

```
p(h1.default)
h1.default = 'A mysterious place'
```

COPYING A HASH

hash2.rb

As with an array, you can assign one Hash variable to another, in which case both variables will refer to the same Hash and a change made using either variable will affect that Hash:

```
h4 = h1
h4['room1']='A new Room'
puts(h1['room1'])          #=> 'A new Room'
```

If you want the two variables to refer to the same items in different Hash objects, use the clone method to make a new copy:

```
h5 = h1.clone
h5['room1'] = 'An even newer Room'
puts(h1['room1'])          #=> 'A new room' (i.e. its value is unchanged)
```

SORTING A HASH

hash_sort.rb

As with the Array class, you may find a slight problem with the sort method of Hash. It expects to be dealing with keys of the same data type so if, for example, you merge two arrays, one of which uses integer keys and another of which uses strings, you won't be able to sort the merged Hash. The solution to this problem is, as with Array, to write some code to perform a custom type of comparison and pass this to the sort method. You might give it a method, like this:

```
def sorted_hash( aHash )
  return aHash.sort{
    |a,b|
      a.to_s <=> b.to_s
  }
end
```

This performs the sort based on the string representation (to_s) of each key in the Hash. In fact, the Hash sort method converts the Hash to a nested array of *[key, value]* arrays and sorts them using the Array sort method.

HASH METHODS

hash_methods.rb

The Hash class has numerous built-in methods. For example, to delete an item using its key (`someKey`) from a hash, `aHash`, use `aHash.delete( someKey )`. To test if a key or value exists use `aHash.has_key?( someKey )` and `aHash.has_value?( someValue )`. To return a new hash created using the original hash's values as keys, and its keys as values use `aHash.invert`; to return an array populated with the hash's keys or with its values use `aHash.keys` and `aHash.values`, and so on.

The **hash_methods.rb** program demonstrates a number of these methods.

Digging Deeper

TREATING HASHES AS ARRAYS

hash_ops.rb

The keys and values methods of Hash each return an array so you can use various Array methods to manipulate them. Here are a few simple examples:

```
h1 = {'key1'=>'val1', 'key2'=>'val2', 'key3'=>'val3', 'key4'=>'val4'}
h2 = {'key1'=>'val1', 'KEY_TWO'=>'val2', 'key3'=>'VALUE_3',
'key4'=>'val4'}

p( h1.keys & h2.keys )                  # set intersection (keys)
#=> ["key1", "key3", "key4"]

p( h1.values & h2.values )              # set intersection (values)
#=> ["val1", "val2", "val4"]

p( h1.keys+h2.keys )                    # concatenation
#=> [ "key1", "key2", "key3", "key4", "key1", "key3", "key4", "KEY_TWO"]

p( h1.values-h2.values )                # difference
#=> ["val3"]

p( (h1.keys << h2.keys)  )              # append
#=> ["key1", "key2", "key3", "key4", ["key1", "key3", "key4", "KEY_TWO"]]

p( (h1.keys << h2.keys).flatten.reverse  ) # 'un-nest' arrays and reverse
#=> ["KEY_TWO", "key4", "key3", "key1", "key4", "key3", "key2", "key1"]
```

APPENDING AND CONCATENATING

Be careful to note the difference between concatenating using + to add the values from the second array to the first and appending using << to add the second array as the final element of the first:

append_concat.rb

```
a =[1,2,3]
b =[4,5,6]
c = a + b                    #=> c=[1, 2, 3, 4, 5, 6]  a=[1, 2, 3]
a << b                       #=> a=[1, 2, 3, [4, 5, 6]]
```

In addition `<<` modifies the first (the 'receiver') array whereas `+` returns a new array but leaves the receiver array unchanged.

Receivers, Messages and Methods

In Object Oriented terminology, the object to which a method belongs is called the **receiver**. The idea is that instead of 'calling functions' as in procedural languages, 'messages' are sent to objects. For example, the message `+ 1` might be sent to an integer object while the message `reverse` might be sent to a string object. The object which 'receives' a message tries to find a way (that is a '**method**') of responding to the message. A string object, for example, has a `reverse` method so is able to respond to the `reverse` message whereas an integer object has no such method so cannot respond.

If, after appending an array with `<<` you decide that you'd like to add the elements from the appended array to the receiver array rather than have the appended array itself 'nested' inside the receiver, you can do this using the `flatten` method:

```
a=[1, 2, 3, [4, 5, 6]]
a.flatten                    #=> [1, 2, 3, 4, 5, 6]
```

MATRICES AND VECTORS

Ruby provides the Matrix class which may contain rows and columns of values each of which can be represented as a vector (Ruby also supplies a Vector class). Matrices allow you to perform matrix arithmetic. For example, give two Matrix objects, m1 and m2, you can add the values of each corresponding cell in the matrices like this:

matrix.rb

```
m3 = m1+m2
```

SETS

The Set class implements a collection of unordered values with no duplicates. You can initialize a Set with an array of values in which case, duplicates are ignored:

Examples:

sets.rb

```
s1 = Set.new( [1,2,3, 4,5,2] )
s2 = Set.new( [1,1,2,3,4,4,5,1] )
s3 = Set.new( [1,2,100] )
weekdays = Set.new( %w( Monday, Tuesday, Wednesday, Thursday,
              Friday, Saturday, Sunday ) )
```

You can add new values using the add method:

```
s1.add( 1000 )
```

The merge method combines values of one Set with another:

```
s1.merge(s2)
```

You can use == to test for equality. Two sets which contain the same values (remembering that duplicates will be removed when a Set is created) are considered to be equal:

```
p( s1 == s2 )                    #=> true
```

CHAPTER FIVE

Loops and Iterators

Much of programming is concerned with repetition. Maybe you want your program to beep ten times, read lines from a file just so long as there are more lines to read or display a warning until the user presses a key. Ruby provides a number of ways of performing this kind of repetition.

FOR LOOPS

In many programming languages, when you want to run a bit of code a certain number of times you can just put it inside a `for` loop. In most languages, you give a `for` loop a variable initialized with a starting value which is incremented by 1 on each turn through the loop until it meets some specific ending value. When the ending value is met, the `for` loop stops running. Here's a version of this traditional type of `for` loop written in Pascal:

```
(* This is Pascal code, not Ruby! *)
for i := 1 to 3 do
  writeln( i );
```

for_loop.rb

You may recall from the last chapter that Ruby's `for` loop doesn't work like this at all! Instead of giving it a starting and ending value, we give the `for` loop a list of items and it iterates over them, one by one, assigning each value in turn to a loop variable until it gets to the end of the list.

For example, here is a for loop that iterates over the items in an array, displaying each in turn:

```
# This is Ruby code…
for i in [1,2,3] do
  puts( i )
end
```

The for loop is more like the 'for each' iterator provided by some other programming languages. The items over which the loop iterates don't have to be integers. This works just as well…

```
for s in ['one','two','three'] do
  puts( s )
end
```

The author of Ruby describes for as "syntax sugar" for the each method which is implemented by collection types such as Arrays, Sets, Hashes and Strings (a String being, in effect, a collection of characters). For the sake of comparison, this is one of the for loops shown above rewritten using the each method:

each_loop.rb

```
[1,2,3].each  do |i|
  puts( i )
end
```

As you can see, there isn't really all that much difference. To convert the for loop to an each iterator, all I've had to do is delete for and in and append .each to the array. Then I've put the iterator variable, i, between a pair of upright bars after do. Compare these other examples to see just how similar for loops are to each iterators:

for_each.rb

```
# --- Example 1 ---
# i) for
for s in ['one','two','three'] do
  puts( s )
end
```

```
# ii) each
['one','two','three'].each do |s|
  puts( s )
end

# --- Example 2 ---
# i) for
for x in [1, "two", [3,4,5] ] do puts( x ) end

# ii) each
[1, "two", [3,4,5] ].each do |x| puts( x ) end
```

Note, incidentally, that the `do` keyword is optional in a `for` loop that spans multiple lines but it is obligatory when it is written on a single line:

```
# Here the 'do' keyword can be omitted
for s in ['one','two','three']
  puts( s )
end

# But here it is required
for s in ['one','two','three'] do puts( s ) end
```

for_to.rb

How to write a 'normal' for loop…

If you miss the traditional type of `for` loop, you can always 'fake' it in Ruby by using a `for` loop to iterate over the values in a range. For example, this is how to use a `for` loop variable to count up from 1 to 10, displaying its value at each turn through the loop:

```
for i in (1..10) do
  puts( i )
end
```

for_each2.rb

This example shows how both for and each can be used to iterate over the values in a range:

```
# for
for s in 1..3
  puts( s )
end

# each
(1..3).each do |s|
  puts(s)
end
```

Note, incidentally, that a range expression such as 1..3 must be enclosed between round brackets when used with the each method, otherwise Ruby assumes that you are attempting to use each as a method of the final integer (a Fixnum) rather than of the entire expression (a Range). The brackets are optional when a range is used in a for loop.

MULTIPLE ITERATOR ARGUMENTS

multi_array.rb

You may recall that in the last chapter we used a for loop with more than one loop variable. We did this in order to iterate over a multi-dimensional array. On each turn through the for loop, a variable was assigned one row (that is, one 'sub-array') from the outer array:

```
# Here multiarr is an array containing two 'rows'
# (sub-arrays) at index 0 and 1
multiarr = [          ['one','two','three','four'],
            [1,2,3,4]
          ]
```

```
# This for loop runs twice (once for each 'row' of multiarr)
for (a,b,c,d) in multiarr
  print("a=#{a}, b=#{b}, c=#{c}, d=#{d}\n" )
end
```

The above loop prints this:

```
a=one, b=two, c=three, d=four
a=1, b=2, c=3, d=4
```

We could use the `each` method to iterate over this four-item array by passing four 'block parameters' - `a, b, c, d` – into the block delimited by `do` and `end` at each iteration:

```
multiarr.each do |a,b,c,d|
  print("a=#{a}, b=#{b}, c=#{c}, d=#{d}\n" )
end
```

Block Parameters

In Ruby the body of an iterator is called a 'block' and any variables declared between upright bars at the top of a block are called 'block parameters'. In a way, a block works like a function and the block parameters work like a function's argument list. The `each` method runs the code inside the block and passes to it the arguments supplied by a collection (such as the array, `multiarr`). In the example above, the `each` method repeatedly passes an array of four elements to the block and those elements initialize the four block parameters, `a, b, c, d`. Blocks can be used for other things, in addition to iterating over collections. I'll have more to say on blocks in Chapter 10.

BLOCKS

block_syntax.rb

Ruby has an alternative syntax for delimiting blocks. Instead of using do..end, you can use curly braces {..} like this:

```
# do..end
[[1,2,3],[3,4,5],[6,7,8]].each do
  |a,b,c|
    puts( "#{a}, #{b}, #{c}" )
end

# curly braces {..}
[[1,2,3],[3,4,5],[6,7,8]].each{
  |a,b,c|
    puts( "#{a}, #{b}, #{c}" )
}
```

No matter which block delimiters you use, you must ensure that the opening delimiter, '{' or 'do', is placed on the same line as the each method. Inserting a line break between each and the opening block delimiter is a syntax error.

WHILE LOOPS

Ruby has a few other loop constructs too. This is how to do a while loop:

```
while tired
  sleep
end
```

Or, to put it another way:

```
sleep while tired
```

Even though the syntax of these two examples is different they perform the same function. In the first example, the code between while and end (here a call to a method named sleep) executes just as long as the Boolean condition (which, in this case, is the value returned by a method called tired) evaluates to true. As in for loops the keyword do may optionally be placed between the test condition and the code to be executed when these appear on separate lines; the do keyword is obligatory when the test condition and the code to be executed appear on the same line.

WHILE MODIFIERS

In the second version of the loop (sleep while tired), the code to be executed (sleep) precedes the test condition (while tired). This syntax is called a 'while modifier'. When you want to execute several expressions using this syntax, you can put them between the begin and end keywords:

```
begin
  sleep
  snore
end while tired
```

1loops.rb

This is an example showing the various alternative syntaxes:

```
$hours_asleep = 0

def tired
  if $hours_asleep >= 8 then
    $hours_asleep = 0
    return false
  else
    $hours_asleep += 1
    return true
  end
end

def snore
  puts('snore....')
end
```

```
def sleep
  puts("z" * $hours_asleep )
end

while tired do sleep end          # a single-line while loop

while tired                       # a multi-line while loop
  sleep
end

sleep while tired                 # single-line while modifier

begin                             # multi-line while modifier
  sleep
  snore
end while tired
```

The last example above (the multi-line `while` modifier) needs close consideration as it introduces some important new behaviour. When a block of code delimited by `begin` and `end` precedes the `while` test, that code always executes at least once. In the other types of `while` loop, the code may never execute at all if the Boolean condition initially evaluates to true.

Ensuring a Loop Executes At Least Once

Usually a `while` loops executes 0 or more times since the Boolean test is evaluated *before* the loop executes; if the test returns false at the outset, the code inside the loop never runs.

However, when the `while` test follows a block of code enclosed between `begin` and `end`, the loop executes 1 or more times as the Boolean expression is evaluated *after* the code inside the loop executes.

2loops.rb

To appreciate the differences in behaviour of these two types of while loop, run **2loops.rb**.

These examples should help to clarify:

```
x = 100

  # The code in this loop never runs
while (x < 100) do puts('x < 100') end

  # The code in this loop never runs
puts('x < 100') while (x < 100)

  # But the code in loop runs once
begin puts('x < 100') end while (x < 100)
```

UNTIL LOOPS

Ruby also has an until loop which can be thought of as a '*while not*' loop. Its syntax and options are the same as those applying to while – that is, the test condition and the code to be executed can be placed on a single line (in which case the do keyword is obligatory) or then can be placed on separate lines (in which case do is optional).

There is also an until modifier which lets you put the code before the test condition and an option to enclose the code between begin and end in order to ensure that the code block is run at least once.

until.rb

Here are some simple examples of until loops:

```
i = 10

until i == 10 do puts(i) end # never executes

until i == 10                    # never executes
  puts(i)
  i += 1
end

puts(i) until i == 10            # never executes

begin                            # executes once
  puts(i)
end until i == 10
```

Both while and until loops can, just like a for loop, be used to iterate over arrays and other collections. For example, this is how to iterate over all the elements in an array:

```
while i < arr.length
  puts(arr[i])
  i += 1
end

until i == arr.length
  puts(arr[i])
  i +=1
end
```

LOOP

3loops.rb

The examples in **3loops.rb** should all look pretty familiar – with the exception of the last one:

```
loop {
  puts(arr[i])
  i+=1
  if (i == arr.length) then
    break
  end
}
```

This uses the loop method repeatedly to execute the block enclosed by curly braces. This is just like the iterator blocks we used earlier with the each method. Once again, we have a choice of block delimiters – either curly braces or do and end:

```
puts( "\nloop" )
i=0
loop do
  puts(arr[i])
  i+=1
  if (i == arr.length) then
    break
  end
end
```

This code iterates through the array, arr, by incrementing a counter variable, i, and breaking out of the loop when the (i == arr.length) condition evaluates to true. You have to break out of a loop in this way since, unlike while or until, the loop method does not evaluate a test condition to determine whether or not to continue looping. Without a break it would loop forever.

Digging Deeper

Hashes, Arrays, Ranges and Sets all include a Ruby module called Enumerable. A module is a sort of code library (I'll have more to say about modules in Chapter 12). In Chapter 4, I used the Comparable module to add comparison methods such as < and > to an array. You may recall that I did this by subclassing the Array class and 'including' the Comparable module into the subclass:

```
class Array2 < Array
  include Comparable
end
```

THE ENUMERABLE MODULE

enum.rb

The Enumerable module is already included into the Ruby Array class and it provides arrays with a number of useful methods such as `include?` which returns true if a specific value is found in an array, `min` which returns the smallest value, `max` which returns the largest and `collect` which creates a new array made up of values returned from a block:

```
arr = [1,2,3,4,5]
y = arr.collect{ |i| i }       #=> y = [1, 2, 3, 4]
z = arr.collect{ |i| i * i }   #=> z = [1, 4, 9, 16, 25]

arr.include?( 3 )              #=> true
arr.include?( 6 )              #=> false
arr.min                        #=> 1
arr.max                        #=> 5
```

enum2.rb

These same methods are available to other collection classes just as long as those classes include Enumerable. Hash is such a class. Remember, however, that the items in a Hash are not indexed in sequential order so when you use the `min` and `max` methods these return the items that are lowest and highest according to their numerical value – here the items are strings and the numerical value is determined by the ASCII codes of the characters in the key.

CUSTOM COMPARISONS

But let's suppose you would prefer min and max to return items based on some other criterion (say the length of a string)? The easiest way to do this would be to define the nature of the comparison inside a block. This is done in a similar manner to the sorting blocks I defined in Chapter 4. You may recall that we sorted a Hash (here the variable h) by passing a block to the sort method like this:

```
h.sort{ |a,b| a.to_s <=> b.to_s  }
```

The two parameters, a and b, represent two items from the Hash which are compared using the <=> comparison method. We can similarly pass blocks to the max and min methods:

```
h.min{ |a,b| a[0].length <=> b[0].length }
h.max{|a,b| a[0].length <=> b[0].length }
```

When a Hash passes items into a block it does so in the form of arrays, each of which contains a key-value pair. So, if a Hash contains items like this…

```
{'one'=>'for sorrow', 'two'=>'for joy'}
```

…the two block arguments, a and b would be initialized to two arrays:

```
a = ['one','for sorrow']
b = ['two','for joy']
```

This explains why the two blocks in which I have defined custom comparisons for the max and min methods specifically compare the first elements, at index 0, of the two block parameters:

```
a[0].length <=> b[0].length
```

This ensures that the comparisons are based on the *keys* in the Hash.

If you want to compare the *values* rather than the keys, just set the array indexes to 1:

enum3.rb

```
p( h.min{|a,b| a[1].length <=> b[1].length } )
p( h.max{|a,b| a[1].length <=> b[1].length } )
```

You could, of course, define other types of custom comparisons in your blocks. Let's suppose, for example, that you want the strings 'one', 'two', 'three' and so on, to be evaluated in the order in which we would speak them. One way of doing this would be to create an ordered array of strings:

```
str_arr=['one','two','three','four','five','six','seven']
```

Now, if a Hash, `h`, contains these strings as keys, a block can use `str_array` as a reference in order to determine the minimum and maximum values:

```
h.min{|a,b| str_arr.index(a[0]) <=> str_arr.index(b[0])}
      #=> ["one", "for sorrow"]

h.max{|a,b| str_arr.index(a[0]) <=> str_arr.index(b[0])}
      #=> ["seven", "for a secret never to be told"]
```

All the examples above, use the `min` and `max` methods of the Array and Hash classes. Remember that these methods are provided to those classes by the Enumerable module.

There may be occasions when it would be useful to be able to apply Enumerable methods such as `max`, `min` and `collect` to classes which do not descend from existing classes (such as Array) which implement those methods. You can do that by including the Enumerable module in your class and then writing an iterator method called `each` like this:

include_enum1.rb

```
class MyCollection
  include Enumerable

  def initialize( someItems )
   @items = someItems
  end

  def each
   @items.each{ |i|
    yield( i )
   }
  end
end
```

Here you could initialize a MyCollection object with an array, which will be stored in the instance variable, `@items`. When you call one of the methods provided by the Enumerable module (such as `min`, `max` or `collect`) this will, 'behind the scenes', call the `each` method in order to obtain each piece of data one at a time.

Now you can use the Enumerable methods with your MyCollection objects:

```
things = MyCollection.new(['x','yz','defgh','ij','klmno'])

p( things.min )                         #=> "defgh"
p( things.max )                         #=> "yz"
p( things.collect{ |i| i.upcase } )
                              #=> ["X", "YZ", "DEFGH", "IJ", "KLMNO"]
```

include_enum2.rb

You could similarly use your MyCollection class to process arrays such as the keys or values of Hashes. Currently the `min` and `max` methods adopt the default behaviour of performing comparisons based on numerical values so 'xy' will be considered to be 'higher' than 'abcd' on the basis of the characters' ASCII values. If you want to perform some other type of comparison – say, by string length, so that 'abcd' would be deemed to be higher than 'xz' - you can just override the min and max methods:

include_enum3.rb

```
def min
  @items.to_a.min{|a,b| a.length <=> b.length }
end

def max
  @items.to_a.max{|a,b| a.length <=> b.length }
end
```

Each and Yield…

So what is really going on when a method from the Enumerable module makes use of the each method that you've written? It turns out that the Enumerable methods (min, max, collect and so forth) pass to the each method a block of code. This block of code expects to receive one piece of data at a time (namely each item from a collection of some sort). Your each method supplies it with that item in the form of a block parameter, such as the parameter i here:

```
def each
   @items.each{ |i|
      yield( i )
   }
end
```

The keyword yield is a special bit of Ruby magic which here tells the code to run the block that was passed to the each method – that is, to run the code supplied by the Enumerator module's min, max or collect methods. This means that the code of those methods can be used with all kinds of different types of collection. All you have to do is, i) include the Enumerable module into your class and ii) write an each method which determines which values will be used by the Enumerable methods.

CHAPTER SIX

Conditional Statements

Computer programs, like Life Itself, are full of difficult decisions waiting to be made. Things like: *If I stay in bed I will get more sleep, else I will have to go to work; if I go to work I will earn some money, else I will lose my job* - and so on…

We've already performed a number of `if` tests in previous programs. To take a simple example, this is from the Tax calculator in chapter one:

```
if (subtotal < 0.0) then
  subtotal = 0.0
end
```

In this program, the user was prompted to enter a value, `subtotal`, which was then used in order to calculate the tax due on it. If the user, in a fit of madness, enters a value less than 0, the `if` test spots this since the test `(subtotal < 0.0)` evaluates to true, which causes the body of the code between the `if` test and the `end` keyword to be executed; here, this sets the value of `subtotal` to 0.

Equals once = or equals twice == ?

In common with many other programming languages, Ruby uses *one* equals sign to assign a value `=` and *two* to test a value `==`.

IF..THEN..ELSE

if_else.rb

A simple test like this has only one of two possible results. Either a bit of code is run or it isn't, depending on whether the test evaluates to true or not. Often, you will need to have more than two possible outcomes. Let's suppose, for example, that your program needs to follow one course of action if the day is a weekday and a different course of action if it is a weekend. You can test these conditions by adding an `else` section after the `if` section, like this:

```
if aDay == 'Saturday' or aDay == 'Sunday'
  daytype = 'weekend'
else
  daytype = 'weekday'
end
```

The `if` condition here is straightforward. It tests two possible conditions: 1) if the value of the variable `aDay` is equal to the string 'Saturday' or 2) if the value of `aDay` is equal to the string 'Sunday'. If either of those conditions is true then the next line of code executes: `daytype = 'weekend'`; in all other cases, the code after `else` executes: `daytype = 'weekday'`.

if_then.rb

When an `if` test and the code to be executed are placed on separate lines, the `then` keyword is optional. When the test and the code are placed on a single line, the `then` keyword (or, if you prefer really terse code, a colon character) is obligatory:

```
if x == 1 then puts( 'ok' ) end          # with 'then'

if x == 1 : puts( 'ok' ) end             # with colon

if x == 1 puts( 'ok' ) end               # syntax error!
```

An `if` test isn't restricted to evaluating just two conditions. Let's suppose, for example, that your code needs to work out whether a certain day is a working day or a holiday. All weekdays are working days; all Saturdays are holidays but Sundays are only holidays when you are not working overtime. This is my first attempt to write a test to evaluate all these conditions:

and_or_wrong.rb

```
working_overtime = true
if aDay == 'Saturday' or aDay == 'Sunday' and not working_overtime
  daytype = 'holiday'
  puts( "Hurrah!" )
else
  daytype = 'working day'
end
```

Unfortunately, this doesn't have quite the effect intended. Remember that Saturday is always a holiday. But this code insists that 'Saturday' is a working day. This is because Ruby takes the test to mean: "If the day is Saturday and I am not working overtime, or if the day is Sunday and I am not working overtime" whereas what I really meant was "If the day is Saturday; or if the day is Sunday and I am not working overtime". The easiest way to resolve this ambiguity is to put brackets around any code to be evaluated as a single unit, like this:

and_or.rb

```
if aDay == 'Saturday' or (aDay == 'Sunday' and not working_overtime)
```

AND..OR..NOT

Incidentally, Ruby has two different syntaxes for testing Boolean (true/false) conditions. In the above example, I've used the English-language style operators: `and`, `or` and `not`. If you prefer you could use alternative operators similar to those used in many other programming languages, namely: `&&` (and), `||` (or) and `!` (not).

Be careful, though, the two sets of operators aren't completely interchangeable. For one thing, they have different precedence which means that when multiple operators are used in a single test the parts of the test may be evaluated in different orders depending on which operators you use. For example, look at this test:

days.rb

```
if aDay == 'Saturday' or aDay == 'Sunday' and not working_overtime
  daytype = 'holiday'
end
```

Assuming that the Boolean variable, working_overtime, is true, would this test succeed if the variable, aDay, were initialised with the string, 'Saturday'? In other words, would daytype be assigned the value 'holiday' if aDay is 'Saturday'? The answer is: no, it wouldn't. The test will only success if aDay is either 'Saturday' or 'Sunday' and working_overtime is not true.

Now consider this test:

```
if aDay == 'Saturday' || aDay == 'Sunday' &&  !working_overtime
  daytype = 'holiday'
end
```

On the face of it, this is the same test as the last one; the only difference being that this time I've used the alternative syntax for the operators. However, the change is more than cosmetic since, if aDay is 'Saturday' this test evaluates to true and daytype is initialized with the value, 'holiday'. This is because the || operator has a higher precedence than the or operator. So this test succeeds either if aDay is 'Saturday' *or* if aDay is 'Sunday' and working_overtime is not true.

Refer to the **Digging Deeper** section at the end of this chapter for more on this. As a general principle, you would do well to decide which set of operators you prefer, stick to them and use brackets to avoid ambiguity.

IF..ELSIF

There will no doubt be occasions when you will need to take multiple different actions based on several alternative conditions. One way of doing this is by evaluating one if condition followed by a series of other test conditions placed after the keyword elsif. The whole lot must then be terminated using the end keyword.

For example, here I am repeatedly taking input from a user inside a while loop; an if condition tests if the user enters 'q' (I've used chomp() to remove the carriage return from the input); if 'q' is not entered the first elsif condition tests if the integer value of the input (input.to_i) is greater than 800; if this test fails the next elsif condition tests if it is less than or equal to 800:

if_elsif.rb

```
while input != 'q' do
  puts("Enter a number between 1 and 1000 (or 'q' to quit)")
  print("?- ")
  input = gets().chomp()
  if input == 'q'
    puts( "Bye" )
  elsif input.to_i > 800
    puts( "That's a high rate of pay!" )
  elsif input.to_i <= 800
    puts( "We can afford that" )
  end
end
```

The problem with this program is that, even though it asks the user to enter a value between 1 and 1000, it accepts values less than 1 (incidentally, if you really want a salary in minus figures, I'll be glad to offer you a job!) and greater than 1000 (in which case, don't look to me for employment!).

We can fix this by rewriting the two elsif conditions and adding an else section which executes if all the preceding tests fail:

if_elsif2.rb

```
if input == 'q'
  puts( "Bye" )
elsif input.to_i > 800 && input.to_i <= 1000
  puts( "That's a high rate of pay!" )
elsif input.to_i <= 800 && input.to_i > 0
  puts( "We can afford that" )
else
  puts( "I said: Enter a number between 1 and 1000!" )
end
```

if_else_alt.rb

Ruby also has a short-form notation for if..then..else in which a question mark ? replaces the if..then part and a colon : acts as else…

< Test Condition > ? <if true do this> : <else do this>

For example:

```
x == 10 ? puts("it's 10") : puts( "it's some other number" )
```

When the test condition is complex (if it uses ands and ors) you should enclose it in brackets. If the tests and code span several lines the ? must be placed on the same line as the preceding condition and the : must be placed on the same line as the code immediately following the ?. In other words, if you put a newline before the ? or the : you will generate a syntax error. This is an example of a valid multi-line code block:

```
(aDay == 'Saturday' or aDay == 'Sunday') ?
  daytype = 'weekend' :
  daytype = 'weekday'
```

days2.rb

Here's another example of a longer sequence of if..elsif sections followed by a catch-all else section. This time the trigger value, i, is an integer:

```
def showDay( i )
  if i == 1 then puts("It's Monday" )
  elsif i == 2 then puts("It's Tuesday" )
  elsif i == 3 then puts("It's Wednesday" )
  elsif i == 4 then puts("It's Thursday" )
  elsif i == 5 then puts("It's Friday" )
  elsif (6..7) === i then puts( "Yippee! It's the weekend! " )
  else puts( "That's not a real day!" )
  end
end
```

Notice that I've used a range (6..7) to match the two integer values for Saturday and Sunday. The === method (that is, three = characters) tests whether a value (here i) is a member of the range. In the above example, this…

```
(6..7) === i
```

…could be rewritten as:

```
(6..7).include?(i)
```

The === method is defined by the Object class and overridden in descendent classes. Its behaviour varies according to the class. As we shall see shortly, one of its fundamental uses is to provide meaningful tests for case statements.

UNLESS

unless.rb

Ruby also can also perform unless tests, which are the exact opposite of if tests:

```
unless aDay == 'Saturday' or aDay == 'Sunday'
  daytype = 'weekday'
else
  daytype = 'weekend'
end
```

Think of unless as being an alternative way of expressing 'if not'. The following is equivalent to the code above:

```
if !(aDay == 'Saturday' or aDay == 'Sunday')
  daytype = 'weekday'
else
  daytype = 'weekend'
end
```

IF AND UNLESS MODIFIERS

You may recall the alternative syntax for while loops mentioned in Chapter 5. Instead of writing this…

```
while tired do sleep end
```

…we can write this:

```
sleep while tired
```

This alternative syntax, in which the while keyword is placed between the code to execute and the test condition is called a 'while modifier'. It turns out that Ruby has if and unless modifiers too. Here are a few examples:

if_unless_mod.rb

```
sleep if tired

begin
  sleep
  snore
end if tired

sleep unless not tired

begin
  sleep
  snore
end unless not tired
```

The terseness of this syntax is useful when you repeatedly need to take some well-defined action if some condition is true. You might, for example, pepper your code with debugging output if a constant called DEBUG is true:

```
puts( "somevar = #{somevar}" ) if DEBUG
```

constants.rb

Constants

Constants in Ruby begin with a capital letter. Class names are constants. You can obtain a list of all defined constants using the `constants` method:

```
Object.constants
```

Ruby provides the `const_get` and `const_set` methods to get and set the value of named constants specified as symbols (identifiers preceded by a colon such as `:RUBY_VERSION`).

Note that, unlike the constants in many other programming languages, Ruby's constants may be assigned new values:

```
RUBY_VERSION = "1.8.7"
RUBY_VERSION = "2.5.6"
```

The above reassignment of the `RUBY_VERSION` constant produces an 'already initialized constant' warning – but not an error!

CASE STATEMENTS

When you need to take a variety of different actions based on the value of a single variable, multiple `if..elsif` tests are verbose and repetitive.

A neater alternative is provided by a `case` statement. This begins with the word `case` followed by the variable name to test. Then comes a series of `when` sections, each of which specifies a 'trigger' value followed by some code.

This code executes only when the test variable equals the trigger value:

case.rb

```
case( i )
  when 1 : puts("It's Monday" )
  when 2 : puts("It's Tuesday" )
  when 3 : puts("It's Wednesday" )
  when 4 : puts("It's Thursday" )
  when 5 : puts("It's Friday" )
  when (6..7) : puts( "Yippee! It's the weekend! " )
  else puts( "That's not a real day!" )
end
```

In the example above, I've used colons to separate each `when` test from the code to execute. Alternatively, you could use the `then` keyword:

```
when 1 then puts("It's Monday" )
```

The colon or `then` can be omitted if the test and the code to be executed are on separate lines. Unlike `case` statements in C-like languages, there is no need to enter a `break` keyword when a match is made in order to prevent execution trickling down through the remainder of the sections. In Ruby, once a match is made the `case` statement exits:

```
case( i )
  when 5 : puts("It's Friday" )
    puts("...nearly the weekend!")
  when 6 : puts("It's Saturday!" )
      # the following never executes
  when 5 : puts( "It's Friday all over again!" )
end
```

You can include several lines of code between each `when` condition and you can include multiple values separated by commas to trigger a single when block, like this:

```
when 6, 7 : puts( "Yippee! It's the weekend! " )
```

case2.rb

The condition in a `case` statement is not obliged to be a simple variable; it can be an expression like this:

```
case( i + 1 )
```

You can also use non-integer types such as string. If multiple trigger values are specified in a `when` section, they may be of varying types – for example, both string and integers:

```
when 1, 'Monday', 'Mon' : puts( "Yup, '#{i}' is Monday" )
```

Here is a longer example, illustrating some of the syntactical elements mentioned above:

case3.rb

```
case( i )
  when 1 : puts("It's Monday" )
  when 2 : puts("It's Tuesday" )
  when 3 : puts("It's Wednesday" )
  when 4 : puts("It's Thursday" )
  when 5 then puts("It's Friday" )
    puts("...nearly the weekend!")
  when 6, 7
    puts("It's Saturday!" ) if i == 6
    puts("It's Sunday!" ) if i == 7
    puts( "Yippee! It's the weekend! " )
     # the following never executes
  when 5 : puts( "It's Friday all over again!" )
  else puts( "That's not a real day!" )
end
```

THE === METHOD

As mentioned earlier, the when tests on object used in a case statement are performed using the === method. So, for example, just as the === method returns true when an integer forms part of a range, so a when test returns true when an integer variable in a case statement forms part of a range expression:

```
when (6..7) : puts( "Yippee! It's the weekend! " )
```

If in doubt on the effect of the === method for a specific object, refer to the Ruby documentation on that object's class.

ALTERNATIVE CASE SYNTAX

There is an alternative form of the case statement which is like a shorthand form of a series of if..then..else statements. Each when section can perform some arbitrary test and execute one or more lines of code. No case variable is required. Each when section returns a value which, just like a method, is the result of the last piece of code that's evaluated. This value can be assigned to a variable preceding the case statement:

case4.rb

```
salary = 2000000
season = 'summer'

happy = case
  when salary > 10000 && season == 'summer':
     puts( "Yes, I really am happy!" )
       'Very happy'                  #=> This value is 'returned'
  when salary > 500000 && season == 'spring' : 'Pretty happy'
  else puts( 'miserable' )
end

puts( happy ) #=> "Very happy"
```

Digging Deeper

BOOLEAN CONDITIONS

`and &&`

These operators evaluate the left-hand side then, only if the result is true, they evaluate the right-hand side; `and` has lower precedence than `&&`

`or   ||`

These operators evaluate the left-hand side then, if the result is false, they evaluate the right-hand side; `or` has lower precedence than `||`

`not !`

This negates a Boolean value – i.e. returns true when false and false when true.

Be careful when using the alternative Boolean operators. Due to the difference in precedence, conditions will be evaluated in different orders and may yield different results.

Consider the following:

boolean_ops.rb

```
# Example 1
if ( 1==3 ) and (2==1) || (3==3) then
  puts('true')
else
  puts('false')
end

# Example 2
if ( 1==3 ) and (2==1) or (3==3) then
  puts('true')
else
  puts('false')
end
```

These may look the same at first sight. In fact, Example 1 prints 'false' while example 2, prints true. This is entirely due to the fact that `or` has lower precedence than `||`. As a consequence, *Example 1* tests: "if 1 equals 3 [*false*] and (either 2 equals 1 or 3 equals 3) [*true*]". As one of these two necessary conditions is false, the entire test returns false.

Now look at *Example 2*. This tests: "(if 1 equals 3 and 2 equals 1) [*false*] or 3 equals 3 [*true*]". This time, we only need one of the two tests to succeed; the second test evaluates to true so the entire tests returns true.

The side-effects of operator precedence in this kind of test can lead to very obscure bugs. You can avoid these by clarifying the meaning of the test using brackets. Here, I have rewritten Examples 1 and 2 above; in each case the addition of one pair of brackets has inverted the initial Boolean value returned by the test:

```
# Example 1 (b) – now returns true
if (( 1==3 ) and (2==1)) || (3==3) then
  puts('true')
else
  puts('false')
end

# Example 2 (b) – now returns false
if ( 1==3 ) and ((2==1) or (3==3)) then
  puts('true')
else
  puts('false')
end
```

NEGATION

The negation operator, !, can be used at the start of an expression or, as an alternative, you can use the != ('not equals') operator between the left and right hand side of an expression:

```
!(1==1)             #=> false
1!=1                #=> false
```

Alternatively, you can use **not** instead of !

```
not( 1==1 )
```

ECCENTRICITIES OF BOOLEAN OPERATORS

eccentricities.rb

Be warned that Ruby's Boolean operators can sometimes behave in a curious and unpredictable manner. For example…

```
puts( (not( 1==1 )) )              # This is ok
puts( not( 1==1 ) )                # This is a syntax error

puts( true && true && !(true) )    # This is ok
puts( true && true and !(true) )   # This is a syntax error

puts( ((true) and (true)) )                  # This is ok
puts( true && true )               # This is ok
puts( true and true )              # This is a syntax error
```

In many cases, problems can be avoided by sticking to one style of operator (that is – either and, or, not *or* &&, ||, !) rather than mixing the two. In addition, the generous use of brackets is to be recommended!

CATCH AND THROW

Ruby provides a pair of methods, catch and throw, which can be used to break out of a block of code when some condition is met. This is Ruby's nearest equivalent to a goto in some other programming languages. The block must begin with catch followed by a symbol (i.e. a unique identifier preceded by a colon), such as :done or :finished. The block itself may either be delimited by curly brackets or by the keywords do and end, like this:

```
# think of this as a block called :done
catch( :done ){
  # some code here
}

# and this is a block called :finished
catch( :finished ) do
  # some code here
end
```

Inside the block, you can call `throw` with a symbol as an argument. Normally you would call `throw` when some specific condition is met which makes it desirable to skip all the remaining code in the block. For instance, let's assume the block contains some code that prompts the user to enter a number, divides some value by that number then goes on to do a multitude of other complex calculations with the result. Obviously, if the user enters 0 then none of the calculations that follow can be completed so you would want to skip them all by jumping right out of the block and continuing with any code that follows it. This is one way of doing that:

catch_throw.rb

```
catch( :finished) do
  print( 'Enter a number: ' )
  num = gets().chomp.to_i
  if num == 0 then
    throw :finished # if num is 0, jump out of the block
  end
    # Here there may be hundreds of lines of
    # calculations based on the value of num
    # if num is 0 this code will be skipped
end
    # the throw method causes execution to
    # jump to here – outside of the block
puts( "Finished" )
```

You can, in fact, have a call to throw outside the block, like this:

```
def dothings( aNum )
  i = 0
  while true
    puts( "I'm doing things..." )
    i += 1
    throw( :go_for_tea ) if (i == aNum )
                              # throws to end of go_to_tea block
  end
end

catch( :go_for_tea ){         # this is the :go_to_tea block
    dothings(5)
}
```

And you can have catch blocks nested inside other catch blocks, like this:

```
catch( :finished) do
  print( 'Enter a number: ' )
  num = gets().chomp.to_i
  if num == 0 then throw :finished end
    puts( 100 / num )


  catch( :go_for_tea ){
    dothings(5)
  }

  puts( "Things have all been done. Time for tea!" )
end
```

As with gotos and jumps in other programming languages, catch and throw in Ruby should be used with great care as they break the logic of your code and can, potentially, introduce hard-to-find bugs.

CHAPTER SEVEN

Methods

We've used numerous methods throughout this book. On the whole, they aren't particularly complicated things – so you may wonder why the present chapter, which is all about methods, is so long. As we shall discover, there is much more to methods than meets the eye.

CLASS METHODS

The methods we've been using so far have been 'instance methods'. An instance method belongs to a specific instance of a class – in other words, to an individual object. It is also possible to write 'class methods'. A class method belongs to the class itself. In order to define a class method you may precede the method name with the class name and a full stop:

class_methods1.rb

```
class MyClass
  def MyClass.classMethod
    puts( "This is a class method" )
  end

  def instanceMethod
    puts( "This is an instance method" )
  end
end
```

You should use the class name when calling a class method:

```
MyClass.classMethod
```

A specific object cannot call a class method. Nor can a class call an instance method:

```
MyClass.instanceMethod         #=> Error! This is an 'undefined method'
ob.classMethod                 #=> Error! This is an 'undefined method'
```

CLASS VARIABLES

Class methods may remind you of the class variables (that is, variables whose names begin with @@). You may recall that we previously used class variables in a simple adventure game (see: **2adventure.rb** in Chapter 2) to keep a tally of the total number of objects in the game; each time a new Thing object was created, 1 was added to the @@num_things class variable:

```
class Thing
  @@num_things = 0

  def initialize( aName, aDescription )
    @@num_things +=1
  end

end
```

Unlike an instance variable (in an object derived from a class), a class variable must be given a value when it is first declared:

```
@@classvar = 1000        # class variables must be initialized
```

Initialization of either instance or class variables within the body of the class only affect the values stored by the class itself. Class variables are available both to the class itself and to objects created from that class. However, each instance variable is unique; each object has its own copy of any instance variables – *and the class itself may also have its own instance variables*.

Class and Instance Variables and Methods: Summary

Instance variables begin with @

```
@myinstvar           # instance variable
```

Class variables begin with @@

```
@@myclassvar         # class variable
```

Instance methods are defined by: def <*MethodName*>

```
def anInstanceMethod
  # some code
end
```

Class methods are defined by: def <*ClassName*>.<*MethodName*>

```
def MyClass.aClassMethod
  # some code
end
```

class_methods2.rb

To understand how a class may have instance variables, take a look at the **class_methods2.rb** program. This declares and initializes a class variable and an instance variable:

```
@@classvar = 1000
@instvar = 1000
```

It defines a class method, classMethod, which increments both these variables by 10 and an instance method, instanceMethod, which increments both variables by 1. Notice that I have also given a value to the instance variable, @instvar. I said earlier that initial values are not normally assigned to instance variables in this way. The exception to the rule is when you assign a value to an instance variable *of the class itself* rather than to an object derived from that class. The distinction should become clearer shortly.

I've written a few lines of code which create three instances of MyClass (the ob variable is initialized with a new instance on each turn through the loop), then calls both the class and instance methods:

```
for i in 0..2 do
  ob = MyClass.new
  MyClass.classMethod
  ob.instanceMethod
  puts( MyClass.showVars )
  puts( ob.showVars )
end
```

I've also written another class method, MyClass.showVars, and an instance method, showVars, to display the values of @instvar and @@classvar at each turn through the loop. When you run the code, these are the values that are displayed:

```
(class method) @instvar = 1010, @@classvar = 1011
(instance method) @instvar = 1, @@classvar = 1011
(class method) @instvar = 1020, @@classvar = 1022
(instance method) @instvar = 1, @@classvar = 1022
(class method) @instvar = 1030, @@classvar = 1033
(instance method) @instvar = 1, @@classvar = 1033
```

You may need to look at these results carefully in order to see what is going on here. In summary, this is what is happening: the code in both the class method, MyClass.classMethod and the instance method, instanceMethod, increments both the class and instance variables, @@classvar and @instvar.

You can see clearly that the class variable is incremented by both these methods (the class method adds 10 to @@classvar whenever a new object is created while the instance method adds 1 to it). However, whenever a new object is created its instance variable is initialized to 1 by the instanceMethod. This is the expected behavior – since each object has its own copy of an instance variable but all objects share a unique class variable.

Perhaps less obvious is the fact that the class itself also has its own instance variable, @instvar. This is because, in Ruby, a class is an object and therefore, can contain instance variables, just like any other object. The MyClass variable, @instvar, is incremented by the class method, MyClass.classMethod:

```
@instvar += 10
```

Notice when the instance method, showVars, prints the value of @instvar, it prints the value stored in a specific object, ob; the value of ob's @instvar is initially nil (*not* the value 1000 with which the MyClass variable, @instvar, was initialized) and this value is incremented by 1 in instanceMethod.

When the class method, MyClass.showVars, prints the value of @instvar, it prints the value stored in the class itself (in other words, MyClass's @instvar *is a different variable* from ob's @instvar). But when either method prints the value of the class variable, @@classvar, the value is the same.

Just remember that there is only ever one copy of a class variable but there may be many copies of instance variables. If this is still confusing, take a look at the **inst_vars.rb** program:

inst_vars.rb

```
class MyClass
  @@classvar = 1000
  @instvar = 1000

  def MyClass.classMethod
    if @instvar == nil then
      @instvar = 10
    else
      @instvar += 10
    end
  end

  def instanceMethod
    if @instvar == nil then
      @instvar = 1
    else
      @instvar += 1
    end
  end
end

ob = MyClass.new
puts MyClass.instance_variable_get(:@instvar)
```

```
puts( '-------------' )
for i in 0..2 do
  # MyClass.classMethod
  ob.instanceMethod
  puts( "MyClass @instvar=#{MyClass.instance_variable_get(:@instvar)}")
  puts( "ob @instvar= #{ob.instance_variable_get(:@instvar)}" )
end
```

This time, instead of creating a new object instance at each turn through the loop, we create a single instance (ob) at the outset. When the ob.instanceMethod is called, @instvar is incremented by 1.

Here I've used a little trick to look inside the class and method and retrieve the value of @instvar using Ruby's instance_get_variable method:

```
puts( "MyClass @instvar= #{MyClass.instance_variable_get(:@instvar)}" )
puts( "ob @instvar= #{ob.instance_variable_get(:@instvar)}" )
```

As we only ever increment the @instvar which belongs to the object ob, the value of its @instvar goes up from 1 to 3 as the for loop executes. But the @instvar which belongs to the MyClass class is never incremented; it remains at its initial value (1000)...

```
1000
-------------
MyClass @instvar= 1000
ob @instvar= 1
MyClass @instvar= 1000
ob @instvar= 2
MyClass @instvar= 1000
ob @instvar= 3
```

But now, uncomment this line...

```
MyClass.classMethod
```

This now calls a class method which increments @instvar by 10. This time when you run the program you see that, as before, the @instvar variable of ob is incremented by 1 on each turn through the loop while the `@instvar` variable of MyClass is incremented by 10...

```
1000
--------------
MyClass @instvar= 1010
ob @instvar= 1
MyClass @instvar= 1020
ob @instvar= 2
MyClass @instvar= 1030
ob @instvar= 3
```

A Class Is An Object

To understand this, just remember that *a class is an object* (actually, it's an instance of the `Class` class!). The MyClass 'class object' has its own instance variable (`@instvar`) just as the `ob` object has *its* own instance variable (which, here, also happens to be called `@instvar`). Instance variables are always unique to an object instance – so no two objects (not even an object like MyClass which also happens to be a class!) can ever share a single instance variable.

WHAT ARE CLASS METHODS FOR?

But why, one may reasonably ask, would you ever want to create a class method rather than the more usual instance method? There are two main reasons: first, a class method can be used as a 'ready-to-run function' without having to go to the bother of creating an object just to use it and, secondly, it can be used on those occasions when you need to run a method before an object has been created.

For a few examples of using methods as 'ready to run functions', take a look at the File class. Many of its methods are class methods. This is because, most of the time you will be using them to do something to or return information on an existing file. You don't need to create a File object to do that; instead you pass the file name as an argument to the class methods. Here are a few examples:

file_methods.rb

```
fn = 'file_methods.rb'
if File.exist?(fn) then
  puts(File.expand_path(fn))
  puts(File.basename(fn))
  puts(File.dirname(fn))
  puts(File.extname(fn))
  puts(File.mtime(fn))
  puts("#{File.size(fn)} bytes")
else
  puts( "Can't find file!")
end
```

The other occasion when a class method is vital is when you need to use a method before an object has been created. The most important example of this is the `new` method.

You call the `new` method every time you create an object. Until the object has been created, you clearly cannot call one of its instance methods – because you can only call instance methods from an object that already exists. When you use `new` you are calling a method of the class itself and telling the class to create a new instance of itself.

RUBY CONSTRUCTORS – NEW OR INITIALIZE?

The method responsible for bringing an object into being is called the constructor. In Ruby, the constructor method is called `new`. The `new` method is a class method which, once it has created an object, will run an instance method named `initialize` if such a method exits.

In brief then, the `new` method is the constructor and the `initialize` method is used to initialize the values of any variables immediately after an object is created. But why can't you just write your own `new` method and initialize variables in it? Well, let's try that:

new.rb

```
class MyClass
  def initialize( aStr )
    @avar = aStr
  end

  def MyClass.new( aStr )
    super
    @anewvar = aStr.swapcase
  end
end

ob = MyClass.new( "hello world" )
puts( ob )
puts( ob.class )
```

Here, I've called the default new constructor using the super keyword to invoke the new method of the super class. Then I've created a string instance variable, @anewvar. So what do I end up with? Not, as you might suppose, a new MyClass object containing a couple of string variables. Remember that the last expression evaluated by a method in Ruby is the value returned by that method. The last expression evaluated by the new method here is a string. So when I evaluate this…

```
ob = MyClass.new( "hello world" )
```

…MyClass.new returns a string; and it is this string (not a MyClass object) which is assigned to ob. As it is most unlikely that you would ever want to do something like this, you would generally be wise to avoid trying to override the new method.

SINGLETON METHODS

class_classes.rb

A singleton method is a method which belong to a single object rather than to an entire class. Many of the methods in the Ruby class library are singleton methods. This is because, as mentioned earlier, each class is an object of the type Class. Or, to put it simply: the class of every class is Class. This is true of all classes – both those you define yourself and those provided by the Ruby class library:

```
class MyClass
end

puts( MyClass.class )      #=> Class
puts( String.class )       #=> Class
puts( Object.class )       #=> Class
puts( Class.class )        #=> Class
puts( IO.class )           #=> Class
```

Now, some classes also have class methods – that is, methods which belong to the Class object itself. In that sense these are singleton methods of the Class object. Indeed, if you evaluate the following, you will be shown an array of method names which match the names of IO class methods:

```
p( IO.singleton_methods )
```

As explained earlier, when you write your own class methods you do so by prefacing the method name with the name of the class:

```
def MyClass.classMethod
```

It turns out that you can use a similar syntax when creating singleton classes for specific objects. This time you preface the method name with the name of the object:

```
def myObject.objectMethod
```

class_hierarchy.rb

All Ruby objects are descendents of the Object class...

...and that includes the Class class! Curious as it may at first seem, each class from which an object is created is itself an object which descends from the Object class. To prove this, try out the **class_hierarchy.rb** program:

```
def showFamily( aClass )
  if (aClass != nil) then
      puts( "#{aClass} :: about to recurse with aClass.superclass
                = #{aClass.superclass}" )
      showFamily( aClass.superclass )
  end
end
```

Let's look at a concrete example. Suppose you have a program containing Creature objects of many different species (maybe you are a veterinarian, the head keeper or a zoo or, like the author of this book, an enthusiastic player of adventure games); each creature has a method called talk which displays the vocal noise which each creature usually makes.

Here's my Creature class and a few creature objects:

singleton_meth1.rb

```
class Creature
  def initialize( aSpeech )
    @speech = aSpeech
  end

  def talk
    puts( @speech )
  end
end

cat = Creature.new( "miaow" )
dog = Creature.new( "woof" )
```

```
budgie = Creature.new( "Who's a pretty boy, then!" )
werewolf = Creature.new( "growl" )
```

Then you suddenly realize that one of those creatures, and one alone, has additional special behavior. On the night of a full moon the werewolf not only talks ("growl"); it also howls ("How-oo-oo-oo-oo!"). It really needs a `howl` method.

You could go back and add such a method to the Creature class but then you'd end up with howling dogs, cats and budgies too – which is not what you want. You could create a new Werewolf class which descends from Creature, but you will only ever have one werewolf (they are, alas, an endangered species) so why do you want a whole class for just that? Wouldn't it make more sense to have a werewolf *object* which is the same as every other creature object apart from the fact that it also has a `howl` method? OK, so let's do that by giving the werewolf its very own singleton method. Here goes:

```
def werewolf.howl
  puts( "How-oo-oo-oo-oo!" )
end
```

Heck, we can do better than that! It only howls on a full moon so let's make sure that, if asked to howl when the moon is new, it just growls. Here's my finished method:

```
def werewolf.howl
   if FULLMOON then
     puts( "How-oo-oo-oo-oo!" )
  else
     talk
  end
end
```

Notice that, even though this method has been declared outside of the Creature class, it is able to call the instance method, `talk`. That's because the `howl` method now lives 'inside' the werewolf object so has the same scope within that object as the `talk` method. It does not, however, live inside any of the werewolf's fellow creatures; the `howl` method belongs to him and him alone. Try to make the `budgie.howl` and Ruby will inform you that `howl` is an undefined method.

Now, if you are debugging your code for your own use, having your program blow up thanks to an undefined method may be acceptable; but if your program does so out in the big, bad world of the 'end user', it is definitely *not* acceptable.

If you think undefined methods are likely to be a problem, you can take avoidance measures by testing if a singleton method exists before trying to use it. The Object class has a singleton_methods method which returns an array of singleton method names. You can test a method name for inclusion using the Array class's include? method. In **singleton_meth2.rb**, for example, I've programmed an 'open the box' game which has a number of Box objects only one of which, when opened, contains the star prize. I've named this special Box object starprize and given it a singleton method called congratulate:

singleton_meth2.rb

```
starprize = Box.new( "Star Prize" )
def starprize.congratulate
  puts( "You've won a fabulous holiday in Grimsby!" )
end
```

The congratulate method should be called when the starprize box is opened. This bit of code (in which item is a Box object) ensures that this method (which does not exist in any other object) is not called when some other box is opened:

```
if item.singleton_methods.include?("congratulate") then
  item.congratulate
end
```

An alternative way of checking the validity of a method would be to pass that method name as a symbol (an identifier preceded by a colon) to the Object class's respond_to? method:

```
if item.respond_to?( :congratulate ) then
  item.congratulate
end
```

We'll look at another way of handling non-existent methods in Chapter 20.

SINGLETON CLASSES

A singleton method is a method which belongs to a single object. A singleton class, on the other hand, is a class which defines a single object. Confused? Me too. So let's take a closer look at the darn' things…

Let's suppose you create a few dozen objects, each of which is an instance of the Object class. Naturally they all have access to the usual Object class's methods such as inspect and class. But now you decide that you want just one special object (for the sake of variety, let's call him ob) which has one special method (let's call it blather).

You don't want to define a whole new class for this one object since you will never again create any more objects with the blather method. So you create a class especially for little ob.

You don't need to name this class. You just tell it to attach itself to ob by putting a << between the keyword class and the name of the object. Then you add code to the class in the usual way:

singleton_class.rb

```
ob = Object.new
     # singleton class
class << ob
  def blather( aStr )
    puts("blather, blather #{aStr}")
  end
end
```

Now ob, and only ob, not only has all the usual methods of the Object class; it also has the methods (here just the blather method but there could, in principle, be many more) of its own special anonymous class:

```
ob.blather( "weeble" )     #=> "blather, blather weeble"
```

If you've been paying close attention, you might have noticed that the singleton class seems to be doing something rather similar to a singleton method. With a singleton class, I can create an object and then add on extra methods packaged up inside an anonymous class. With singleton methods, I can create an object then add on methods one by one:

```
ob2 = Object.new

def ob2.blather( aStr )          # <= this is a singleton method
  puts( "grippity, grippity #{aStr}" )
end

ob2.blather( "ping!" )           #=> grippity, grippity ping!
```

singleton_class2.rb

Similarly, I could rewrite the 'star prize' program. In the previous version I added on a singleton method, `congratulate`, to an object named `starprize`. I could just as easily have created a singleton class containing the `congratulate` method:

```
starprize = MyClass.new( "Star Prize" )

class << starprize
  def congratulate
    puts( "You've won a fabulous holiday in Grimsby!" )
  end
end
```

In fact, the similarity is more than skin deep. The end result of the code above is that congratulate becomes a singleton method of `starprize` and I've been able to verify that using this test:

```
if item.singleton_methods.include?("congratulate")
```

Singleton Method, Singleton Class – What's The Difference...?

The short answer is: not a lot. These two syntaxes provide different ways of adding methods to a specific object rather than building those methods into its defining class.

OVERRIDING METHODS

There are times when you may want to redefine a method that already exists in some class. We've done this before when, for example, we created classes with their own `to_s` methods to return a string representation. Every Ruby class, from Object downwards, has a `to_s` method. The `to_s` method of the Object class returns the class name and a hexadecimal representation of the object's unique identifier. However, many Ruby classes have their own special versions of `to_s`. For example, `Array.to_s` concatenates and returns the values in the array.

When a method in one class replaces a method of the same name in an ancestor class, it is said to 'override' that method. You can override methods that are defined in the standard class library such as `to_s` as well as methods defined in your own classes. If you need to add new behavior to an existing method, remember to call the superclass's method using the `super` keyword at the start of the overridden method. Here is an example:

override.rb

```
class MyClass
  def sayHello
    return "Hello from MyClass"
  end

  def sayGoodbye
    return "Goodbye from MyClass"
  end
end

class MyOtherClass < MyClass
  def sayHello          #overrides (and replaces) MyClass.sayHello
    return "Hello from MyOtherClass"
  end

    # overrides MyClass.sayHello but first calls that method
    # with super. So this version "adds to" MyClass.sayHello
  def sayGoodbye
    return super << " and also from MyOtherClass"
  end
```

```
    # overrides default to_s method
  def to_s
    return "I am an instance of the #{self.class} class"
  end
end
```

PUBLIC, PRIVATE AND PROTECTED

In some cases, you may want to restrict the 'visibility' of your methods to ensure that they cannot be called by code outside the class in which the methods occur.

This may be useful when your class defines various 'utility' methods which it requires in order to perform certain functions which it does not intend for public consumption. By imposing access restrictions on those methods you can prevent programmers from using them for their own nefarious purposes. This means that you will be able to change the implementation of those methods at a later stage without having to worry that you are going to break somebody else's code.

Ruby provides three levels of method accessibility:

```
public
protected
private
```

As the name suggests, **public** methods are the most accessible and **private** methods are the least accessible. All your methods are public unless you specify otherwise. When a method is public, it is available to be used by the world outside the object in whose class it is defined.

When a method is **private**, it can only be used by other methods inside the object in whose class it is defined.

A **protected** method generally works in the same way as a private method with one tiny, but important difference: in addition to being visible to the methods of the current object, a protected method is also visible to objects of the same type when the second object is within the scope of the first object.

The distinction between private and protected methods will probably be easier to understand when you see a working example. Consider this class:

pub_prot_priv.rb

```
class MyClass

  private
    def priv
      puts( "private" )
    end

  protected
    def prot
      puts( "protected" )
    end

  public
    def pub
      puts( "public" )
    end

    def useOb( anOb )
      anOb.pub
      anOb.prot
      anOb.priv
    end
end
```

I've declared three methods, one for each level of accessibility. These levels are set by putting `private`, `protected` or `public` prior to one or more methods. The specified accessibility level remains in force for all subsequent methods until some other access level is specified.

Note: `public`, `private` and `protected` may look like keywords. But they are, in fact, methods of the Module class.

Finally, my class has a public method, `useOb`, which takes a `MyOb` object as an argument and calls the three methods, `pub`, `prot` and `priv` of that object. Now let's see how a MyClass object can be used. First of all, I'll create two instances of the class:

```
myob = MyClass.new
myob2 = MyClass.new
```

Now, I try to call each of the three methods in turn…

```
myob.pub         # This works! Prints out "public"
myob.prot        # This doesn't work! I get a 'NoMethodError'
myob.priv        # This doesn't work either - another 'NoMethodError'
```

From the above, it would seem that the public method is (as expected) visible from the world outside the object to which it applies. But both the private and the protected methods are invisible. This being so, what is the protected method for? Another example should help to clarify this:

```
myob.useOb( myob2 )
```

This time, I am calling the public method useOb of the myob object and I am passing to it a second object, myob2, as an argument. The important thing to note is that myob and myob2 are instances of the same class. Now, recall what I said earlier:

> *in addition to being visible to the methods of the current object, a protected method is also visible to objects of the same type when the second object is within the scope of the first object.*

This may sound like gobbledygook. Let's see if we can start to make some sense out of it.

In the program, the first MyClass object (here myob) has a second MyClass object within its scope when myob2 is passed as an argument to a method of myob. When this happens, you can think of myob2 are being present 'inside' myob. Now myob2 shares the scope of the 'containing' object, myob. In this special circumstance – when two objects of the same class are within the scope defined by that class – the protected methods of any objects of this class become visible.

In the present case, the protected method, prot, of myob2 (or, at any rate, of the argument - here called anob - which 'receives' myob2) becomes visible and can be executed. Its private arguments, however, are not visible:

```
def useOb( anOb )
  anOb.pub
  anOb.prot       # protected method can be called
  anOb.priv       # but calling a private method results in an error
end
```

Digging Deeper

PROTECTED AND PRIVATE IN DESCENDENT CLASSES

The same access rules apply when calling the methods of ancestor and descendent objects. That is, when you pass to a method an object (as an argument) which has the same class as the receiver object (i.e. the object to which the method belongs), the argument object can call the public and protected methods of the class but not its private methods.

protected.rb

For an example of this, take a look at the **protected.rb** program. Here I have created a MyClass object called myob and a MyOtherClass object, myotherob, where MyOtherClass descends from MyClass. I try to pass myotherob as an argument to the myob public method, shout:

```
myob.shout( myotherob )
```

But the shout method calls the private method, priv on the argument object:

```
def shout( anOb )   # calls a private method
  puts( anOb.priv( "This is a #{anOb.class} - hurrah" ) )
end
```

This won't work! Ruby complains that the priv method is private.

Similarly, were I to do it the other way around – that is, by passing the ancestor object, myob, as the argument and invoking the method shout on the descendent object, I would encounter the same error:

```
myotherob.shout( myob )
```

The MyClass class also has another public method, exclaim. This one calls a protected method, prot:

```
def exclaim( anOb ) # calls a protected method
  puts( anOb.prot( "This is a #{anOb.class} - hurrah" ) )
end
```

Now, I can pass either the MyClass object, myob, or the MyOtherClass object, myotherob, as an argument to the exclaim method and no error will occur when the protected method is called:

```
myob.exclaim( myotherob )        # This is OK
myotherob.exclaim( myob )        # And so is this…
```

Needless to say, this only works when the two objects (the receiver and the argument) share the same line of descent. If you send an unrelated object as an argument, you would not be able to call methods of the receiver class, no matter what their protection levels.

INVADING THE PRIVACY OF PRIVATE METHODS

The whole point of a private method is that it cannot be called from outside the scope of the object to which it belongs. So this won't work:

send.rb

```
class X
  private
    def priv( aStr )
      puts("I'm private, " << aStr)
    end
end

ob = X.new
ob.priv( "hello" )     # This fails
```

However, it turns out that Ruby provides a 'get out clause' (or maybe I should say a 'get in' clause?) in the form of a method called send.

The send method invokes the method whose name matches that of a symbol (an identifier beginning with a colon such as :priv), which is passed as the first argument to send like this:

```
ob.send( :priv, "hello" )    # This succeeds
```

Any arguments supplied after the symbol (like the string, "hello") are passed in the normal way to the specified method.

Suffice to say that using **send** to gain public access to a private method is not generally a good idea (else, why would you have made the method private in the first place), so should be used with caution or not at all…

SINGLETON CLASS METHODS

Earlier on, we created class methods by appending a method name to the name of the class like this:

```
def MyClass.classMethod
```

There is a 'shortcut' syntax for doing this. Here is an example:

class_methods3.rb

```
class MyClass

  def MyClass.methodA
    puts("a")
  end

  class << self
    def methodB
      puts("b")
    end

    def methodC
      puts("c")
    end
  end

end
```

Here, **methodA**, **methodB** and **methodC** are all class methods of MyClass; **methodA** is declared using the syntax we used previously:

def <ClassName>.<methodname>

But methodB and methodC are declared using the syntax of instance methods:

def <methodname>

So how come they end up as class methods? It's all down to the fact that the method declarations have been placed inside this code:

```
class << self
  # some method declarations
end
```

This may remind you of the syntax used for declaring singleton classes. For example, in the **singleton_class.rb** program, you may recall that we first created an object named ob and then gave it its very own method, blather:

```
class << ob
  def blather( aStr )
    puts("blather, blather #{aStr}")
  end
end
```

The blather method here is singleton method of the ob object. Similarly, in the **class_methods3.rb** program, the methodB and methodC methods are singleton methods of self – and self happens to be the MyClass class. We can similarly add singleton methods from outside the class definition by using << followed by the class name, like this:

```
class << MyClass
  def methodD
    puts( "d" )
  end
end
```

NESTED METHODS

You can nest methods (have one method nested inside another). This gives you a way of dividing up a long method into reusable chunks. So, for example, if method x needs to do calculation y at several different points, you can put the y method inside the x method:

nested_methods.rb

```
class X

  def x
    print( "x:" )

    def y
      print("ha! ")
    end

    def z
      print( "z:" )
      y
    end

    y
    z
  end

end
```

Nested methods are not initially visible outside of the scope in which they are defined. So, in the above example, while y and z may be called from inside x, they may not be called by any other code:

```
ob = X.new
ob.y        #<= error
ob.z        # <= error
```

However, when you run a method that encloses nested methods, those nested methods *will* be brought into scope outside that method!

nested_methods2.rb

```
class X
  def x
    print( "x:" )
    def y
      print("y:")
    end

    def z
      print( "z:" )
      y
    end
  end
end


ob = X.new
ob.x          #=> x:
puts
ob.y          #=> y:
puts
ob.z          #=> z:y:
```

METHOD NAMES

As a final point, it's worth mentioning that method names in Ruby almost always begin with a lowercase character like this:

```
def fred
```

However, that is a convention, not an obligation. It is also permissible to begin method names with capital letters, like this:

```
def Fred
```

Since the `Fred` method looks like a constant (it starts with a capital letter), you would need to tell Ruby that it is a method when calling it by adding brackets:

method_names.rb

```
Fred        # <= Ruby complains 'uninitialized constant

Fred()      # <= Ruby calls the Fred method
```

On the whole it is better to stick to the convention of using method names that begin with a lowercase character.

CHAPTER EIGHT

Passing Arguments and Returning Values

In this chapter we'll be looking at many of the effects and side-effects of passing arguments and returning values to and from methods. First, though, let's take a moment to summarise the types of method which we've used up to now:

methods.rb

1. INSTANCE METHODS

An instance method is declared inside a class definition and is intended for use by a specific object or 'instance' of the class, like this:

```
class MyClass
  # declare instance method
  def instanceMethod
    puts( "This is an instance method" )
  end
end

  # create object
ob = MyClass.new
  # use instance method
ob.instanceMethod
```

2. CLASS METHODS

A class method may be declared inside a class definition, in which case, a) the method name may be preceded by the class name or b) a `class << self` block may contain a 'normal' method definition; either way, a class method is intended for use by the class itself, not by a specific object, like this:

```
class MyClass
  # a class method
  def MyClass.classmethod1
    puts( "This is a class method" )
  end

  # another class method
  class << self
    def classmethod2
      puts( "This is another class method" )
    end
  end
end

  # call class methods from the class itself
MyClass.classmethod1
MyClass.classmethod2
```

3. SINGLETON METHODS

Singleton methods are methods which are added to a single object and cannot be used by other objects. A singleton method may be defined by appending the method name to the object name followed by a dot or by placing a 'normal' method definition inside an *<ObjectName>* << self block like this:

```
  # create object
ob = MyClass.new

  # define a singleton method
def ob.singleton_method1
  puts( "This is a singleton method" )
end

  # define another singleton method
class << ob
  def singleton_method2
    puts( "This is another singleton method" )
  end
end

  # use the singleton methods
ob.singleton_method1
ob.singleton_method2
```

RETURNING VALUES

In many programming languages, a distinction is made between functions or methods which return a value to the calling code and those which do not. In Pascal, for example, a function returns a value but a procedure does not. No such distinction is made in Ruby. All methods always return a value though you are not, of course, obliged to use it.

return_vals.rb

When no return value is specified, Ruby methods return the result of the last expression evaluated. Consider this method:

```
def method1
  a = 1
  b = 2
  c = a + b   # returns 3
end
```

The last expression evaluated is `a + b` which happens to return 3, so that is the value returned by this method. There may often be times when you don't want to return the last expression evaluated. In such cases, you can specify the return value using the `return` keyword:

```
def method2
  a = 1
  b = 2
  c = a + b
  return b   # returns 2
end
```

A method is not obliged to make any assignments in order to return a value. A simple piece of data (that is, something that evaluates to itself), if this happens to be the last thing evaluated in a method, will be the value returned. When nothing is evaluated, `nil` is returned:

```
def method3
  "hello"   # returns "hello"
end

def method4
  a = 1 + 2
  "goodbye"   # returns "goodbye"
end

def method5
end   # returns nil
```

My own programming prejudice is to write code that is clear and unambiguous whenever possible. For that reason, whenever I plan to use the value returned by a method, I prefer to specify it using the `return` keyword; only when I do not plan to use the returned value do I omit this. However, this is not obligatory – Ruby leaves the choice to you.

RETURNING MULTIPLE VALUES

But what about those occasions when you need a method to return more than one value? In other program languages you may be able to 'fake' this by passing arguments by reference (pointers to the original data items) rather than by value (a copy of the data); when you alter the values of 'by reference' arguments, you alter the original values without explicitly having to return any values to the calling code.

Ruby doesn't make a distinction between 'by reference' and 'by value' so this technique is not available to us (most of the time, anyway, though we shall see some exceptions to the rule shortly). However, Ruby is capable of returning multiple values all in one go, as shown here:

return_many.rb

```
def ret_things
  greeting = "Hello world"
  a = 1
  b = 2.0
  return a, b, 3, "four", greeting, 6 * 10
end
```

Multiple return values are placed into an array. If you were to evaluate `ret_things.class`, Ruby would inform you that the returned object is an Array.

You could, however, explicitly return a different collection type such as a Hash:

```
def ret_hash
  return {'a'=>'hello', 'b'=>'goodbye', 'c'=>'fare thee well'}
end
```

DEFAULT AND MULTIPLE ARGUMENTS

Ruby lets you specify default values for arguments. Default values can be assigned in the parameter list of a method using the usual assignment operator:

```
def aMethod( a=10, b=20 )
```

If an unassigned variable is passed to that method, the default value will be assigned to it. If an assigned variable is passed, however, the assigned value takes precedence over the default:

```
def aMethod( a=10, b=20 )
  return a, b
end

p( aMethod )              #=> displays: [10,  20]
p( aMethod( 1 ))          #=> displays: [1, 20]
p( aMethod( 1, 2 ))       #=> displays: [1, 2]
```

In some cases, a method may need to be capable of receiving an uncertain number of arguments – say, for example, a method which processes a variable length list of items. In this case, you can 'mop up' any number of trailing items by preceding the final argument with an asterisk:

default_args.rb

```
def aMethod( a=10, b=20, c=100, *d )
  return a, b, c, d
end

p( aMethod( 1,2,3,4,6 ) ) #=> displays: [1, 2, 3, [4, 6]]
```

ASSIGNMENT AND PARAMETER PASSING

Most of the time, Ruby methods come with two access points – like the doors into and out of a room. The argument list provides the way in; the return value provides the way out. Modifications made to the input arguments do not affect the original data for the simple reason that, when Ruby evaluates an expression, the result of that evaluation creates a new object – so any changes made to an argument only affect the new object, not the original piece of data. But there are exceptions to this rule, which we'll look at presently.

in_out.rb

Let's start by looking at the simplest case: a method which takes one value as a named parameter and returns another value:

```
def change( x )
  x += 1
  return x
end
```

On the face of it, you might think that we are dealing with a single object, `x`, here: the object, `x`, goes into the `change` method and the same object `x` is returned. In fact, that is not the case. One object goes in (the argument) and a different object comes out (the return value). You can easily verify this using the `object_id` method to show a number which uniquely identifies each object in your program:

```
num = 10
puts( "num.object_id=#{num.object_id}" )
num = change( num )
puts( "num.object_id=#{num.object_id}" )
```

The identifier of the variable, `num`, is different before and after we call the `change` method. This shows that, even though the variable name remains the same, the `num` object which is returned by the `change` method is different from the `num` object which was sent to it.

method_call.rb

The method-call itself has nothing to do with the change of the object. You can verify this by running **method_call.rb**. This simply passes the num object to the change method and returns it:

```
def nochange( x )
  return x
end
```

In this case, the `object_id` is the same after `num` is returned as it was before `num` was sent to the method. In other words, the object that went into the method is exactly the same object as the one that came out again. Which leads to the inevitable conclusion that there is something about the *assignment* in the `change` method (`x += 1`) that caused the creation of a new object.

But assignment itself isn't the whole explanation. If you simply assign a variable to itself, no new object is created…

assignment.rb

```
num = 10
num = num # a new num object is not created
```

So what if you assign to the object the same value which it already has?

```
num = 10
num = 10   # a new num object is not created
```

This demonstrates that assignment alone does not necessarily create a new object. Now let's try assigning a new value…

```
num = 10
num += 1   # this time a new num object is created
```

By checking the `object_id` we are able to determine that when a new value is assigned to an existing variable, a new object is created.

Most data items are treated as unique so one string "hello" is considered to be different from another string "hello" and one float 10.5 is considered to be different from another float 10.5. Thus, any string or float assignment will create a new object.

But when working with integers, only when the assignment value is different from the previous value is a new object created. You can do all kinds of complicated operations on the right-hand part of the assignment but if the yielded value is the same as the original value, no new object is created…

```
num = (((num + 1 - 1) * 100) / 100)        # a new object is not created!
```

object_ids.rb

INTEGERS ARE SPECIAL

In Ruby an integer (Fixnum) has a fixed identity. Every instance of the number 10 or every variable to which the value 10 is assigned will have the same object_id. The same can not be said of other data types. Each instance of a floating point number such as 10.5 or of a string such as "hello world" will be a different object with a unique object_id. Be aware that when you assign an integer to a variable, that variable will have the object_id of the integer itself. But when you assign some other type of data to a variable, a new object will be created even if the data itself is the same at each assignment:

```
# 10 and x after each assignment are the same object
puts( 10.object_id )
x = 10
puts( x.object_id )
x = 10
puts( x.object_id )

# 10.5 and x after each assignment are 3 different objects!
puts( 10.5.object_id )
x = 10.5
puts( x.object_id )
x = 10.5
puts( x.object_id )
```

But why does all this matter?

The answer is that it matters because of a few rare exceptions to the rule. As I said earlier, most of the time, a method has a well-defined way in and a well-defined way out. Once an argument goes inside a method, it enters a closed room. Any code outside that method has no way of learning about any changes that have been made to the argument until it comes out again in the form of a returned value. This is, in fact, one of the deep secrets of 'pure' object orientation. The implementation details of methods should, in principle, be hidden away – 'encapsulated'. This ensures that code outside an object cannot be dependent on things that happen inside that object.

THE ONE-WAY-IN, ONE-WAY-OUT PRINCIPLE

In most modern OOP languages such as Java and C#, encapsulation and information hiding are not rigorously enforced. In Smalltalk, on the other hand - that most famous and influential of OOP languages - encapsulation and information hiding are fundamental principles: if you send a variable, x, to a method y and the value of x is changed inside y, you cannot obtain the changed value of x from outside the method – *unless the method explicitly returns that value.*

'Encapsulation' or 'Information Hiding'?

Often these two terms are used interchangeably. To be nit-picking, however, there is a difference.

Encapsulation refers to the grouping together of an object's 'state' (its data) and the operations which may alter or interrogate its state (its methods).

Information hiding refers to the fact that data is sealed off and can only be accessed using well-defined routes in and out – in object oriented terms this implies 'accessor methods' to get or return values.

In procedural languages, information hiding may take other forms - for example, you might have to define interfaces to retrieve data from code 'units' or 'modules' rather than from objects.

In OOP terms, encapsulation and information hiding are almost synonymous – true encapsulation necessarily implies that the internal data of an object is hidden. However, many modern 'OOP languages' such as Java, C#, C++ and Object Pascal are quite permissive in the degree to which information hiding is enforced (if at all).

Usually, Ruby adheres to this principle: arguments go into a method but any changes made inside the method cannot be accessed from the outside unless Ruby returns the changed value:

hidden.rb

```
def hidden( aStr, anotherStr )
  anotherStr = aStr + " " + anotherStr
  return aStr + anotherStr.reverse
end

str1 = "dlrow"
str2 = "olleh"
str3 = hidden(str1, str2)          # str3 receives returned value
puts( str1 )                       # input args: original values unchanged
puts( str2 )
puts( str3 )                       # returned value ( “dlrowhello world” )
```

It turns out that there are occasions when arguments passed to a Ruby method can be used like the ‘by reference’ arguments of other languages (that is, changes made *inside* the method may affect variables *outside* the method). This is due to the fact that some Ruby methods modify the original object rather than yielding a value and assigning this to a new object.

For example, there are some methods ending with an exclamation mark which alter the original object. Similarly the String append method `<<` concatenates the string on its right to the string on its left but does not create a new string object in the process: so the value of the string on the left is modified but the string object itself retains its original `object_id`.

The consequence of this is that, if you use the `<<` operator instead of the `+` operator in a method, your results will change:

not_hidden.rb

```
def nothidden( aStr, anotherStr )
  anotherStr = aStr << " " << anotherStr
  return aStr << anotherStr.reverse
end

str1 = "dlrow"
str2 = "olleh"
str3 = nothidden(str1, str2)
puts( str1 )                    # input arg: changed ("dlrow ollehhello
world")
puts( str2 )                    # unchanged
puts( str3 )                    # returned value("dlrow ollehhello world")
```

str_reverse.rb

The **str_reverse.rb** sample program should help to clarify this. This shows that when you use the `reverse` method, for example, no change is made to the 'receiver object' (that is, an object such as `str1` here: `str1.reverse`). But when you use the `reverse!` method a change *is* made to the object (its letters are reversed). Even so, no new object is created: `str1` is the same object before and after the `reverse!` method is called.

Here `reverse` operates like most Ruby methods – it yields a value and, in order to use that value, you must assign it to a new object. So...

```
str1 = "hello"
str1.reverse
```

Here, `str1` is unaffected by calling `reverse`. It still has the value "hello" and it still has its original `object_id`. But...

```
str1 = "hello"
str1.reverse!
```

This time, str1 is changed (it becomes "olleh"). Even so, no new object is created: str1 has the same object_id with which it started. Then again...

```
str1 = "hello"
str1 = str1.reverse
```

This time, the value yielded by str1.reverse is assigned to str1. The yielded value is a new object, so str1 is now assigned the reversed string ("olleh") and it now has a new object_id.

Refer to the sample program, **concat.rb**, for an example of the string concatenation method, <<, which, just like those methods that end with !, modifies the receiver object without creating a new object:

concat.rb

```
str1 = "hello"
str2 = "world"
str3 = "goodbye"
str3 = str2 << str1
```

In this example, str1 is never modified so it has the same object_id throughout; str2 *is* modified through concatenation.

However, the << operator does not create a new object so str2 also retains its original object_id. But str3 is a different object at the end than at the beginning: that is because it is assigned the value yielded by this expression: str2 << str1. This value happens to be the str2 object itself so the object_id of str3 is now identical with that of str2 (i.e. str2 and str3 now reference the same object).

In summary, then, methods ending with a ! such as reverse!, plus some other methods such as the << concatenation method, change the value of the receiver object itself. Most other methods do not change the value of the receiver object and in order to make use of any new value yielded as a result of calling a method, you have to assign that value to a variable (or pass the yielded value as an argument to a method).

Modifying The Receiver Object Breaks Encapsulation

The fact that a few methods modify the receiver object whereas most do not may seem harmless enough – but beware: this behaviour provides you with the ability to retrieve the values of arguments 'by reference' rather than retrieving values which are explicitly returned. Doing so breaks encapsulation by allowing your code to rely upon the internal implementation details of a method. This can potentially lead to unpredictable side-effects and, in my view, should be avoided.

side_effects.rb

For a simple (but, in real-world programming, potentially serious) example of how the reliance on the modified values of arguments rather than on explicit return values can introduce undesirable dependencies on implementation details, see **side_effects.rb**. Here we have a method called `stringProcess` which takes two string arguments, messes about with them and returns the results. Let's assume that the object of the exercise is to take two lowercase strings and return a single string which combines these two strings, separated by a space and with the first and last letters capitalized. So the two original strings might be "hello" and "world" and the returned string is "Hello worlD".

But now we have an impatient programmer who can't be bothered with return values. He notices that the modifications made inside the method change the values of the ingoing arguments. So, heck! (he decides) he might as well use the arguments themselves! He then goes away and writes a fabulously complicated text processing system with thousands of bits of code reliant on the changed values of those two arguments.

But now the programmer who originally wrote the `stringProcess` method decides that the original implementation was inefficient or inelegant and so rewrites the code confident in the knowledge that the return value is unchanged (if "hello" and "world" are sent as arguments, "Hello worlD" will be returned).

Aha! But the new implementation cause the values of the input arguments to be changed inside the body of the method. So the impatient programmer's text processing system, which relies on those *arguments* rather than on the return value, is now filled with bits of text saying "hello Dlrow" instead of the "Hello worlD" he was expecting (actually, it turns out that his program was processing the works of Shakespeare so a generation of actors will end up declaiming: "To eb or ton to eb, that si the noitseuq..."). This is the kind of unexpected side-effect which can easily be avoided by following the one-way-in and one-way-out principle…

PARALLEL ASSIGNMENT

I mentioned earlier that it is possible for a method to return multiple values, separated by commas. Often you will want to assign these returned values to a set of matching variables.

In Ruby, this can be done in a single operation by parallel assignment. This means that you can have several variables to the left or an assignment operator and several values to the right. The values to the right will be assigned, in order, to the variables on the left, like this:

parallel_assign.rb

```
s1, s2, s3 = "Hickory", "Dickory", "Dock"
```

This ability not only gives you a shortcut way to make multiple assignments; it also lets you swap the values of variables (you just change their orders on either side of the assignment operator:

```
i1 = 1
i2 = 2

i1, i2 = i2, i1                #=> i1 is now 2, i2 is 1
```

And you can make multiple assignments from the values returned by a method:

```
def returnArray( a, b, c )
  a = "Hello, " + a
  b = "Hi, " + b
  c = "Good day, " + c
  return a, b, c
end
```

```
x, y, z = returnArray( "Fred", "Bert", "Mary" )
```

If you specify more variables to the left than there are values on the right of an assignment, any 'trailing' variables will be assigned `nil`:

```
x, y, z, extravar = returnArray( "Fred", "Bert", "Mary" ) # extravar = nil
```

Multiple values returned by a method are put into an array. When you put an array to the right of a multiple-variable assignment, its individual elements will be assigned to each variable, and once again if too many variables are supplied, the extra ones will be assigned `nil`:

```
s1, s2, s3 = ["Ding", "Dong", "Bell"]
```

Digging Deeper

BY REFERENCE OR BY VALUE?

Search the Internet and you'll soon find that Ruby programmers regularly get into arguments about whether Ruby passes arguments 'by value' or 'by reference'.

In many procedural programming languages such as Pascal and C and their derivatives there is a clear distinction between arguments passed by value or by reference.

A '*by value*' argument is a copy of the original variable; you can pass it to a procedure, mess around with it and the value of the original value remains unchanged.

A '*by reference*' argument, on the other hand, is a pointer to the original variable. When this gets passed to a procedure, you are not passing a new copy but a reference to the bit of memory in which the original data is stored. So any changes made inside the procedure are made to the original data and necessarily affect the value of the original variable.

arg_passing.rb

It's actually pretty easy to resolve this issue. If Ruby passes by value, then it makes a copy of the original variable and that copy will therefore have a different `object_id`. In fact, this is not the case. Try out the **arg_passing.rb** program to prove this point.

Now, it may well be that in certain circumstances the passing of arguments could, 'behind the scenes' so to speak, be *implemented* as 'by value'. However, such implementation details should be of interest to writers of Ruby interpreters and compilers rather than to Ruby programmers. The plain fact of the matter is that, if you program in a 'pure' OOP way – by passing arguments into methods but only subsequently using the values which those methods return – the implementation details (by value or by reference) will be of no consequence to you.

Nevertheless, due to the fact that Ruby can occasionally modify arguments (for example using ! methods or << as explained earlier), some programmers have formed the habit of using the modified values of the arguments themselves (equivalent to using *By Reference* arguments in C) rather than using the values returned. In my view, this is a bad practice. It makes your programs reliant upon the implementation details of methods and should therefore, be avoided.

ARE ASSIGNMENTS COPIES OR REFERENCES?

I said earlier that a new object is created when a value is yielded by some expression. So, for example, if you assign a new value to a variable called x, the object after the assignment will be a different object from the one before the assignment (that is, it will have a different object_id):

```
x = 10                # this x has one object_id
x +=1                 # and this x has a different one
```

But it isn't the assignment that creates a new object. It is the value that is yielded which causes a new object to be created. In the above example, +=1 is an expression that yields a value (x+=1 is equivalent to the expression x=x+1).

Simple assignment of one variable to another does not create a new object. So let's assume you have one variable called num and another called num2. If you assign num2 to num, both variables will refer to the same object. You can test this using the equals? method of the Object class:

assign_ref.rb

```
num = 11.5
num2 = 11.5

    # num and num 2 are not equal
puts( "num.equal?(num2) #{num.equal?(num2)}" )

num = num2
    # but now they are equal
puts( "num.equal?(num2) #{num.equal?(num2)}" )
```

equal_tests.rb

Tests for equality: == or equal?

By default (as defined in Ruby's Kernel module) a test using == returns true when both objects being tested are the same object. So it will return false if the values are same but the objects are different:

```
ob1 = Object.new
ob2 = Object.new
puts( ob1==ob2 ) #<= false
```

In fact == is frequently overridden by classes such as String and will then return true when the values are the same but the objects are different:

```
s1 = "hello"
s2 = "hello"
puts( s1==s2 ) #<= true
```

For that reason, the equal? method is preferable when you want to establish if two variables refer to the same object:

```
puts( s1.equal?(s2) ) #<= false
```

WHEN ARE TWO OBJECTS IDENTICAL?

As a general rule, if you initialize ten variables with ten values, each variable will refer to a different object. For example, if you create two strings like this…

identical.rb

```
s1 = "hello"
s2 = "hello"
```

…then s1 and s2 will refer to independent objects. The same goes for two floats…

```
f1 = 10.00
f2 = 10.00
```

But, as mentioned earlier, integers are different. Create two integers with the same value and they will end up referencing the same object:

```
i1 = 10
i2 = 10
```

This is even true with plain integer values. If in doubt, use the equals? method to test if two variables or values reference exactly the same object:

```
10.0.equal?(10.0)  # compare floats – returns false
10.equal?(10)      # compare integers (Fixnums) – returns true
```

PARENTHESES AVOID AMBIGUITY

Methods may share the same name as a local variable. For example, you might have a variable called `name` and a method called `name`. If it is your habit to call methods without parentheses, it may not be obvious whether you are referring to a method or a variable. Once again, parentheses avoid ambiguity...

parentheses.rb

```
greet = "Hello"
name = "Fred"

def greet
  return "Good morning"
end

def name
  return "Mary"
end

def sayHi( aName )
  return "Hi, #{aName}"
end

puts( greet )                    #<= Hello
puts greet                       #<= Hello
puts( sayHi( name ) )            #<= Hi, Fred
puts( sayHi( name() ) )          #<= Hi, Mary
```

CHAPTER NINE

Exception Handling

Even the most carefully written program will sometimes encounter unforeseen errors. For example, if you write a program that needs to read some data from disk, it works on the assumption that the specified disk is actually available and the data is valid. If your program does calculations based on user input, it works on the assumption that the input is suitable to be used in a calculation.

While you may try to anticipate some potential problems before they arise – for example, by writing code to check that a file exists before reading data from it or checking that user input is numerical before doing a calculation – you will never be able to predict every possible problem in advance.

The user may remove a CD after you've already started reading data from it, for example; or some obscure calculation may yield 0 just before your code attempts to divide by this value. When you know that there is the possibility that your code may be 'broken' by some unforeseen circumstances at runtime, you can attempt to avoid disaster by using 'exception handling'.

An 'exception' is an error which is packaged up into an object. The object is an instance of the Exception class (or one of its descendents). You can handle exceptions by trapping the Exception object, optionally using information which it contains (to print an appropriate error message, say) and taking any actions needed to recover from the error – perhaps by closing any files that are still open or assigning a sensible value to a variable which may have been assigned some nonsensical value as the result of an erroneous calculation.

RESCUE

The basic syntax of exception handling can be summarised as follows:

```
begin
  # Some code which may cause an exception
rescue <Exception Class>
  # Code to recover from the exception
end
```

Here is an example of an exception handler which deals with an attempt to divide by zero:

exception1.rb

```
begin
  x = 1/0
rescue Exception
  x = 0
  puts( $!.class )
  puts( $! )
end
```

div_by_zero.rb

When this code is run, the attempt to divide by zero causes an exception. If unhandled (as in the sample program, **div_by_zero.rb**), the program will crash. However, by placing the troublesome code inside an exception handling block (between `begin` and `end`), I have been able to trap the Exception in the section beginning with `rescue`. The first thing I've done is to set the variable, `x`, to a meaningful value. Next come these two inscrutable statements:

```
puts( $!.class )
puts( $! )
```

In Ruby, `$!` is a global variable to which is assigned the last exception. Printing `$!.class` displays the class name, which here is "ZeroDivisionError"; printing the variable `$!` alone has the effect of displaying the error message contained by the Exception object which here is "divided by 0".

I am not generally too keen on relying upon global variables, particularly when they have 'names' as undescriptive as $!. Fortunately, there is an alternative. You can associate a variable name with the exception by placing the 'assoc operator', =>, after the class name of the exception and before the variable name:

exception2.rb

```
rescue Exception => exc
```

You can now use the variable name (here exc) to refer to the Exception object:

```
puts( exc.class )
puts( exc )
```

exception_tree.rb

Exceptions Have A Family Tree…

To understand how rescue clauses trap exceptions, just remember that, in Ruby, exceptions are objects and, like all other objects, they are defined by a class. There is, moreover, a clear 'line of descent' which starts, like all Ruby objects, with the Object class.

While it may seem pretty obvious that, when you divide by zero, you are going to get a ZeroDivisionError exception, in real world code, there may be times when the type of exception is not so predictable. Let's suppose for instance, that you have a method which does a division based on two values supplied by a user:

```
def calc( val1, val2 )
  return val1 / val2
end
```

This could potentially produce a variety of different exceptions. Obviously if the second value entered by the user is 0, we will get a ZeroDivisionError.

However, if the *second* value is a string, the exception will be a TypeError, whereas is the *first* value is a string it will be a NoMethodError (as the String Class does not define the 'division operator' /). Here the rescue block handles all possible exceptions:

multi_except.rb

```
def calc( val1, val2 )
  begin
    result = val1 / val2
  rescue Exception => e
    puts( e.class )
    puts( e )
    result = nil
  end
  return result
end
```

Often it will be useful to take different actions for different exceptions. You can do that by adding multiple `rescue` clauses. Each `rescue` clause can handle multiple exception types, with the exception class names separated by commas. Here my `calc` method handles TypeError and NoMethodError exceptions in one clause with a catch-all Exception handler to deal with other exception types:

multi_except2.rb

```
def calc( val1, val2 )
  begin
    result = val1 / val2
  rescue TypeError, NoMethodError => e
    puts( e.class )
    puts( e )
    puts( "One of the values is not a number!" )
    result = nil
  rescue Exception => e
    puts( e.class )
    puts( e )
    result = nil
  end
  return result
end
```

exception_tree.rb

The Object class is the ultimate ancestor of all exceptions.

From Object, descends Exception, then StandardError and finally more specific types of exception such as ZeroDivisionError. You could, if you wished, write a rescue clause to deal with the Object class and, Object being the ancestor of all objects, this would, indeed, successfully match an exception object:

```
# This is possible...
rescue Object => exc
```

However, it is generally more useful to try to match a relevant descendent of the Exception class. For good measure, it is often useful to append a generalized rescue clause to handle StandardError or Exception objects, just in case an exception which you hadn't thought of manages to slip through. You may want to run the **exception_tree.rb** program to view the family tree of the ZeroDivisionError exception.

When handling multiple exception types you should always put the rescue clauses dealing with specific exceptions first, then follow these with rescue clauses dealing with more generalized exceptions.

When a specific exception, such as TypeError, is handled, the begin..end exception block exits so the flow of execution won't 'trickle down' to more generalized rescue clauses. However, if you put a generalized exception handling rescue clause first, that will handle all exceptions so any more specific clauses lower down will never execute.

If, for example, I had reversed the order of the rescue clauses in my calc method, placing the generalized Exception handler first, this would match all exception types so the clause for the specific TypeError and NoMethodError exceptions would never be run:

multi_except_err.rb

```
  # This is incorrect...
  rescue Exception => e
      puts( e.class )
      puts( e )
      result = nil
    rescue TypeError, NoMethodError => e
      puts( e.class )
      puts( e )
      puts( "Oops! This message will never be displayed!" )
      result = nil
    end
```

ENSURE

There may be some circumstances in which you want to take some particular action whether or not an exception occurs. For example, whenever you are dealing with some kind of unpredictable input/output – say, when working with files and directories on disk – there is always the possibility that the location (the disk or directory) or the data source (the file) either may not be there at all or may provide some other kinds of problems – such the disk being full when you attempt to write to it or the file containing the wrong kind of data when you attempt to read from it.

You may need to perform some final 'cleanup' procedures – such as logging onto a specific working directory or closing a file which was previously opened - whether or not you have encountered any problems. You can do this by following a `begin..rescue` block of code with another block starting with the `ensure` keyword. The code in the `ensure` block will always execute – whether or not an exception has arisen beforehand.

Let's look at two simple examples. In the first one, I try to log onto a disk and display the directory listing. At the end of this, I want to be sure that my working directory (given by `Dir.getwd`) is always restored to its original location. I do this by saving the original directory in the `startdir` variable and once again making this the working directory in the `ensure` block:

ensure.rb

```
startdir = Dir.getwd

begin
  Dir.chdir( "X:\\" )
  puts( `dir` )
rescue Exception => e
  puts e.class
  puts e
ensure
  Dir.chdir( startdir )
end
```

Let's now see how to deal with the problem of reading the incorrect data from a file. This might happen if the data is corrupt, if you accidentally open the wrong file or – quite simply – if your program code contains a bug.

Here I have a file, **test.txt**, containing six lines. The first five lines are numbers; the sixth line is not. My code opens this file and reads in all six lines:

ensure2.rb

```
f = File.new( "test.txt" )
begin
  for i in (1..6) do
    puts("line number: #{f.lineno}")
    line = f.gets.chomp
    num = line.to_i
    puts( "Line '#{line}' is converted to #{num}" )
    puts( 100 / num )
  end
rescue Exception => e
  puts( e.class )
  puts( e )
ensure
  f.close
  puts( "File closed" )
end
```

The lines are read in as strings (using gets) and the code attempts to convert them to integers (using to_i). No error is produced when the conversion fails; instead Ruby returns the value 0.

The problem arises in the next line of code which attempts a division by the converted number. The sixth line of the input file contains the string "six" which yields 0 when a conversion to integer is attempted – and that inevitably causes an error when this value is used in a division.

Having opened the data file at the outset, I want to ensure that the file is closed whether or not an error occurs. If, for example, I only read in the first five lines by editing the range in the for loop to (1..5), then there would be no exception. I would still want to close the file.

But it would be no good putting the file closing code (f.close) in the rescue clause as it would not, in this case, be executed. By putting it in the ensure clause, however, I can be certain that the file will be closed whether or not an exception occurs.

ELSE

If the rescue section executes when an error occurs and ensure executes whether or not an error occurs, how can we specifically execute some code only when an error does *not* occur?

The way to do this is to add an optional else clause after the rescue section and before the ensure section (if there is one), like this:

```
begin
            # code which may cause an exception
rescue [Exception Type]
else        # optional section executes if no exception occurs
ensure      # optional exception always executes
end
```

This is an example:

else.rb

```
def doCalc( aNum )
  begin
    result = 100 / aNum.to_i
  rescue Exception => e                 # executes when there is an error
    result = 0
    msg = "Error: " + e
  else                                  # executes when there is no error
    msg = "Result = #{result}"
  ensure                                # always executes
    msg = "You entered '#{aNum}'. " + msg
  end
  return msg
end
```

ERROR NUMBERS

If you ran the **ensure.rb** program earlier and you were watching closely you may have noticed something unusual when you tried to log onto a non-existent drive (for example, on my system that might be the "X:\" drive). Often, when an exception occurs, the exception class is an instance of a specific named type such as ZeroDivisionError or NoMethodError. In this case, however, the class of the exception is shown to be:

```
Errno::ENOENT
```

It turns out that there is quite a variety of `Errno` errors in Ruby. Try out **disk_err.rb**. This defines a method, `chDisk`, which attempts to log onto a disk identified by the character, `aChar`. So if you pass "A" as an argument to `chDisk` it will try to log onto the A:\ drive. I've called the `chDisk` method three times, passing to it a different string each time:

disk_err.rb

```
chDisk( "D" )
chDisk( "X" )
chDisk( "ABC" )
```

On my PC, D:\ is my DVD drive. At the moment it is empty and when my program tries to log onto it, Ruby returns an exception of this type:

```
Errno::EACCES
```

I have no X:\ drive on my PC and when I try to log onto that, Ruby returns an exception of this type:

```
Errno::ENOENT
```

In the last example, I pass a string parameter, "ABC" which is invalid as a disk identifier, and Ruby returns an exception of this type:

```
Errno::EINVAL
```

Errors of this type are descendents of the SystemCallError class. You can easily verify this by uncommenting the line of code to show the class's family where indicated in the source code of **disk_err.rb**.

These classes, in effect, wrap up integer error values which are returned by the underlying operating system. Here `Errno` is the name of the module containing the constants, such as `EACCES` and `ENOENT`, which match the integer error values.

To see a complete list of `Errno` constants, run this:

```
puts( Errno.constants )
```

To view the corresponding numerical value of any given constant, append `::Errno` to the constant name, like this:

```
Errno::EINVAL::Errno
```

errno.rb

The following code can be used to display a list of all Errno constants along with their numerical values:

```
for err in Errno.constants do
  errnum = eval( "Errno::#{err}::Errno" )
  puts( "#{err}, #{errnum}" )
end
```

RETRY

If you think an error condition may be transient or may be corrected (by the user, perhaps?), you can rerun all the code in a begin..end block using the keyword retry, as in this example which prompts the user to re-enter a value if an error such as ZeroDivisionError occurs:

retry.rb

```
def doCalc
  begin
    print( "Enter a number: " )
    aNum = gets().chomp()
    result = 100 / aNum.to_i
  rescue Exception => e
    result = 0
    puts( "Error: " + e + "\nPlease try again." )
    retry                    # retry on exception
  else
    msg = "Result = #{result}"
  ensure
    msg = "You entered '#{aNum}'. " + msg
  end
  return msg
end
```

There is, of course, the danger that the error may not be as transient as you think so, if you use retry, you may want to provide a clearly defined exit condition to ensure that the code stops executing after a fixed number of attempts.

You could, for example, increment a local variable in the begin clause (if you do this, make sure it is incremented *before* any code that is liable to generate an exception since, once an exception occurs, the remainder of the code prior to the rescue clause will be skipped!). Then test the value of that variable in the rescue section, like this:

```
rescue Exception => e
    if aValue < someValue then
      retry
    end
```

Here is a complete example, in which I test the value of a variable named tries to ensure that no more than three tries to run the code without error before the exception-handling block exits:

retry2.rb

```
def doCalc
  tries = 0
  begin
    print( "Enter a number: " )
    tries += 1
    aNum = gets().chomp()
    result = 100 / aNum.to_i
  rescue Exception => e
    msg = "Error: " + e
    puts( msg )
    puts( "tries = #{tries}" )
    result = 0
    if tries < 3 then                # set a fixed number of retries
      retry
    end
  else
    msg = "Result = #{result}"
  ensure
    msg = "You entered '#{aNum}'. " + msg
  end
  return msg
end
```

RAISE

Sometimes you may want to keep an exception 'alive' even after it has been trapped in an exception-handling block. This can be used, for example, to defer the handling of the exception – say, by passing it on to some other method. You can do this using the raise method. You need to be aware, however, that once raised, an exception needs to be re-handled otherwise it may cause your program to crash. Here is a simple example of raising a ZeroDivisionError exception and passing on the exception to a method called, in this case, handleError:

raise.rb

```
begin
  divbyzero
rescue Exception => e
  puts( "A problem just occurred. Please wait..." )
  x = 0
  begin
    raise
  rescue
    handleError( e )
  end
end
```

Here divbyzero is the name of a method in which the divide-by-zero operation takes place and handleError is a method that prints more detailed information on the exception:

```
def handleError( e )
  puts( "Error of type: #{e.class}" )
  puts( e )
  puts( "Here is a backtrace: " )
  puts( e.backtrace )
end
```

Notice that this uses the backtrace method which displays an array of strings showing the file names and line numbers where the error occurred and, in this case, the line which called the error-producing divbyzero method.

raise2.rb

You can also specifically raise your exceptions to force en error condition even when the program code itself has not caused an exception. Calling raise on its own raises an exception of the type RuntimeError (or whatever exception is in the global variable $!):

```
raise                    # raises RuntimeError
```

By default, this will have no descriptive message associated with it. You can add a message as a parameter, like this:

```
raise "An unknown exception just occurred!"
```

You may raise a specific type of error…

```
raise ZeroDivisionError
```

You may also create an object of a specific exception type and initialize it with a custom message…

```
raise ZeroDivisionError.new( "I'm afraid you divided by Zero" )
```

raise3.rb

If the standard exception types don't meet your requirements, you can, of course, create new ones just by subclassing existing exceptions. Provide your classes with a to_str method in order to give them a default message.

```
class NoNameError < Exception
  def to_str
    "No Name given!"
  end
end
```

And this is an example of how you might raise a custom exception:

```
def sayHello( aName )
  begin
    if (aName == "") or (aName == nil) then
      raise NoNameError
    end
  rescue Exception => e
    puts( e.class )
    puts( "message: " + e )
    puts( e.backtrace )
  else
    puts( "Hello #{aName}" )
  end
end
```

Digging Deeper

OMITTING BEGIN AND END

You may optionally omit begin and end when trapping exceptions inside a method, a class or a module. For example, all the following are legal:

omit_begin_end.rb

```
def calc
    result = 1/0
  rescue Exception => e
    puts( e.class )
    puts( e )
    result = nil
  return result
end

class X
    @@x = 1/0
  rescue Exception => e
    puts( e.class )
    puts( e )
end

module Y
    @@x = 1/0
  rescue Exception => e
    puts( e.class )
    puts( e )
end
```

In all the cases shown above, the exception-handling will also work if you place the begin and end keywords at the start and end of the exception-handling code in the usual way.

CATCH...THROW

In some languages, exceptions are trapped using the keyword catch and may be raised using the keyword throw. While Ruby provides catch and throw methods, these are not directly related to its exception handling. Instead, catch and throw are used to break out of a defined block of code when some condition is met. You could, of course, use catch and throw to break out of a block of code when an exception occurs (though this may not be the most elegant way of handling errors). For example, this code will exit the block delimited by curly brackets if a ZeroDivisionError occurs:

catch_except.rb

```
catch( :finished) {
  print( 'Enter a number: ' )
  num = gets().chomp.to_i
  begin
    result = 100 / num
  rescue Exception => e
    throw :finished              # jump to end of block
  end
  puts( "The result of that calculation is #{result}" )
}    # end of :finished catch block
```

See Chapter 6 for more on catch and throw.

CHAPTER TEN

Blocks, Procs and Lambdas

When programmers talk about 'blocks', they usually mean some arbitrary 'chunks' of code. In Ruby, however, a block is special. It is a unit of code that works somewhat like a method but, unlike a method, it has no name. In order to use blocks effectively, you need to understand how and why they are special. That is what this chapter is all about...

WHAT IS A BLOCK?

Consider this code:

1blocks.rb

```
3.times do |i|
   puts( i )
end
```

It's probably pretty obvious that this code is intended to execute three times. What may less obvious is the value which i will have on each successive turn through the loop. In fact, the values of i in this case will be 0, 1, and 2. Here is an alternative form of the code above. This time the block is delimited by curly brackets rather than by do and end:

```
3.times { |i|
   puts( i )
}
```

According to the Ruby documentation, times is a method of Integer (let's call the Integer int), which iterates a block "int times, passing in values from zero to int – 1". So, here, the code within the block is run 3 times; the first time it is run the variable, i, is given the value 0; each subsequent time, i is incremented by 1 until the final value, 2 (i.e. int-1) is reached.

Note that the two code examples above are functionally identical. A block may be enclosed either by curly brackets or by the do and end keywords and the programmer may user either syntax according to personal preference.

Note: Some Ruby programmers like to delimit blocks with curly brackets when the entire code of the block fits onto a single line and with do..end when the block spans multiple lines. My personal prejudice is to be consistent, irrespective of code layout, and so I generally use curly braces when delimiting blocks. Usually your choice of delimiters make no difference to the behaviour of the code - but see the section later in this chapter on 'precedence rules'.

If you are familiar with a C-like language such as C# or Java, you may, perhaps, assume that Ruby's curly braces can be used, as in those languages, simply to group together arbitrary 'blocks' of code – for example, a block of code to be executed when a condition evaluates to true. This is not the case. In Ruby, a block is a special construct which can only be used in very specific circumstances.

LINE BREAKS ARE SIGNIFICANT

The opening block delimiter must be placed on the same line as the method with which it is associated.

These are ok...

```
3.times do |i|
   puts( i )
end

3.times { |i|
   puts( i )
}
```

But these contain syntax errors...

```
3.times
do |i|
   puts( i )
end

3.times
{ |i|
   puts( i )
}
```

NAMELESS FUNCTIONS

A Ruby block may be regarded as a sort of nameless function or method and its most frequent use is to provides a means of iterating over items from a list or range of values. If you have never come across nameless functions previously, this may sound like gobbledygook. With luck, by the end of this chapter, things will have become a little clearer. Let's look back at the simple example given earlier. I said a block is like a nameless function. Take this block as an example:

```
{ |i|
   puts( i )
}
```

If that were written as a normal Ruby method it would look something like this:

```
def aMethod( i )
  puts( i )
end
```

To call that method three times and pass values from 0 to 2 we might write this:

```
for i in 0..2
  aMethod( i )
end
```

When you create a nameless method (that is, a block) variables declared between upright bars such as |i| can be treated like the arguments to a named method. We shall refer to these variables as 'block parameters'.

Look again at my earlier example:

```
3.times { |i|
   puts( i )
}
```

The times method of an integer passes values to a block from 0 to the specified integer value minus 1.

So this:

```
3.times{ |i| }
```

…is very much like this:

```
for i in 0..2
  aMethod( i )
end
```

The chief difference is that the second example has to call some other named method to process the value of i whereas the first example uses the nameless method (the code between curly braces) to process i.

LOOKS FAMILIAR?

Now that you know what a block is, you may notice that you've seen them before. Many times.

For example, we previously used do..end blocks to iterate over ranges like this:

```
(1..3).each do |i|
  puts(i)
end
```

We have also used do..end blocks to iterate over arrays (see **for_each2.rb** in Chapter 5):

```
arr = ['one','two','three','four']
arr.each do |s|
  puts(s)
end
```

And we have executed a block repeatedly by passing it to the loop method (see **3loops.rb** in Chapter 5):

```
i=0
loop {
  puts(arr[i])
  i+=1
  if (i == arr.length) then
    break
  end
}
```

The loop example above is notable for two things: 1) It has no list of items (such as an array or a range of values) to iterate over and 2) it is pretty darn' ugly. These two features are not entirely unrelated! The loop method is part of the Kernel class, which is 'automatically' available to your programs. As it has no 'end value' it will execute the block for ever unless you explicitly break out of it using the break keyword. Usually there are more elegant ways to perform this kind of iteration – by iterating over a sequence of values with a finite range.

BLOCKS AND ARRAYS

Blocks are commonly used to iterate over arrays. The Array class, consequently, provides a number of methods to which blocks are passed.

One useful method is called collect; this passes each element of the array to a block and creates a new array to contain each of the values returned by the block. Here, for example, a block is passed each of the integers in an array (each integer is assigned to the variable, x), it doubles its value and returns it.

The collect method creates a new array containing each of the returned integers in sequence:

2blocks.rb

```
b3 = [1,2,3].collect{|x| x*2}
```

The example above returns this array: [2,4,6].

In this next example, the block returns a version of the original strings in which each initial letter is capitalized:

```
b4 = ["hello","good day","how do you do"].collect{|x| x.capitalize }
```

So b4 is now...

```
["Hello", "Good day", "How do you do"]
```

The each method of the Array class may look rather similar to collect; it too passes each array element in turn to be processed by the block. However, unlike collect, the each method does not create a new array to contain the returned values:

```
b5 = ["hello","good day","how do you do"].each{|x| x.capitalize }
```

This time, b5 is unchanged...

```
["hello", "good day", "how do you do"]
```

Recall, however that some methods – notably those ending with an exclamation mark (!) – actually alter the original objects rather than yielding new values. If you wanted to use the each method to capitalize the strings in the original array, you could use the capitalize! method:

```
b6 = ["hello","good day","how do you do"].each{|x| x.capitalize! }
```

So b6 is now...

```
["Hello", "Good day", "How do you do"]
```

With a bit of thought, you could also use a block to iterate over the characters in a string. First, you need to split off each character from a string. This can be done using the split method of the String class like this:

```
"hello world".split(//)
```

The split method divides a string into substrings based on a delimiter and returns an array of these substrings. Here // is a regular expression that defines a zero-length string; this has the effect of returning a single character, so we end up creating an array of all the characters in the string. We can now iterate over this array of characters, returning a capitalized version of each:

```
a = "hello world".split(//).each{ |x| newstr << x.capitalize }
```

So, at each iteration, a capitalized character is appended to newstr, and the following is displayed...

```
H
HE
HEL
HELL
HELLO
HELLO
HELLO W
HELLO WO
HELLO WOR
HELLO WORL
HELLO WORLD
```

As we are using the capitalize method here (with no terminating ! character), the characters in the array, a, remain as they began, all lowercase, since the capitalize method does not alter the receiver object (here the receiver objects are the characters passed into the block).

Be aware, however, that this code would not work if you were to use the capitalize! method to modify the original characters. This is because capitalize! returns nil when no changes are made so when the space character is encountered nil would be returned and our attempt to append (<<) a nil value to the string, newstr, would fail.

You could also capitalize a string using the each_byte method. This iterates through the string characters, passing each byte to the block. These bytes take the form of ASCII codes. So "hello world" would be passed in the form of these numeric values: **104 101 108 108 111 32 119 111 114 108 100**

Obviously, you can't capitalize an integer so we need to convert each ASCII value to a character. The chr method of String does this:

```
a = "hello world".each_byte{|x| newstr << (x.chr).capitalize }
```

PROCS AND LAMBDAS

In our examples up to now, blocks have been used in cahoots with methods. This has been a requirement since nameless blocks cannot have an independent existence in Ruby. You cannot, for example, create a standalone block like this:

```
{|x| x = x*10; puts(x)}
```

This is one of the exceptions to the rule that 'everything in Ruby is an object'. A block clearly is not an object. Every object is created from a class and you can find an object's class by calling its class method.

Do this with a Hash, for example and the class name, 'Hash' will be displayed:

```
puts({1=>2}.class)
```

Try this with a block, however, and you will only get an error message:

```
puts({|i| puts(i)}.class) #<= error!
```

Block Or Hash?

Ruby uses curly brackets to delimit both blocks and Hashes. So how can you (and Ruby) tell which is which? The answer, basically, is that it's a Hash when it *looks* like a Hash, otherwise it's a block. A Hash looks like a Hash when curly brackets contain key-value pairs...

```
puts( {1=>2}.class )      #<= Hash
```

...or when they are empty:

```
puts( {}.class )          #<= Hash
```

However, once again, if you omit the brackets, there is an ambiguity. Is this an empty Hash or is it a block associated with the `puts` method?

```
puts{}.class
```

Frankly, I have to admit that I don't know the answer to that question and I can't get Ruby to tell me. Ruby accepts this as valid syntax but does not, in fact, display anything when the code executes. While, this...

```
print{}.class
```

...prints `nil` (not, you will notice the actual class of `nil`, which is Nil-Class, but `nil` itself). If you find all this confusing (as I do!) just remember that this can all be clarified by the judicious use of brackets:

```
print( {}.class ) #<= Hash
```

CREATING OBJECTS FROM BLOCKS

proc_create.rb

While blocks may not be objects by default, they can be 'turned into' objects. There are three ways of creating objects from blocks and assigning them to variables – here's how:

```
a = Proc.new{|x| x = x*10; puts(x) }
b = lambda{|x| x = x*10; puts(x) }
c = proc{|x| x.capitalize! }
```

Note that, in each of the three cases above, you will end up creating an instance of the Proc class – which is the Ruby 'object wrapper' for a block.

Let's take a closer look at the three ways of creating a Proc object. First, you can create an object calling Proc.new and passing to it a block as an argument:

3blocks.rb

```
a = Proc.new{|x| x = x*10; puts(x)}
```

You can execute the code in the block to which a refers using the Proc class's call method with one or more arguments (matching the block parameters) to be passed into the block; in the code above, you could pass an integer such as 100 and this would be assigned to the block variable, x:

```
a.call(100)
```

You can also create a Proc object by calling the lambda or proc methods. These methods (supplied by the Kernel class) are identical. The name lambda is taken from the Scheme (Lisp) language and is a term used to describe an anonymous method or 'closure'.

There is one important difference between creating a Proc object using Proc.new and creating a Proc object using the proc or lambda methods – Proc.new does not check that the number or arguments passed to the block match the number of block parameters – both proc and lambda do:

proc_lamba.rb

```
a = Proc.new{|x,y,z| x = y*z; puts(x) }
a.call(2,5,10,100)          # This is not an error

b = lambda{|x,y,z| x = y*z; puts(x) }
b.call(2,5,10,100)          # This is an error

puts('---Block #2---' )
c = proc{|x,y,z| x = y*z; puts(x) }
c.call(2,5,10,100)          # This is an error
```

block_closure.rb

WHAT IS A CLOSURE?

'Closure' is the name given to a function which has the ability to store (that is, to 'enclose') values of local variables within the scope in which the block was created (think of this as the block's 'native scope'). Ruby's blocks are closures. To understand this, look at this example:

```
x = "hello world"

ablock = Proc.new { puts( x ) }

def aMethod( aBlockArg )
  x = "goodbye"
  aBlockArg.call
end

puts( x )
ablock.call
aMethod( ablock )
ablock.call
puts( x )
```

Here, the value of the local variable, `x` is "hello world" within the scope of `ablock`. Inside `aMethod`, however, a local variable named `x` has the value "goodbye". In spite of that, when ablock is passed to aMethod and called within the scope of aMethod, it prints "hello world" (that is, the value of x within the block's 'native scope' rather "goodbye" which is the value of x within the scope of aMethod. The above code, therefore, only ever prints "hello world".

See **Digging Deeper** at the end of this chapter for more on closures.

YIELD

Let's see a few more blocks in use. The **4blocks.rb** program introduces something new – namely, a way of executing a nameless block when it is passed to a method. This is done using the keyword yield. In the first example, I define this simple method:

4blocks.rb

```
def aMethod
  yield
end
```

It doesn't really have any code of its own. Instead, it expects to receive a block and the yield keyword causes the block to execute. This is how I pass a block to it:

```
aMethod{ puts( "Good morning" ) }
```

Notice that this time the block is not passed as a named argument. It would be an error to try to pass the block between round brackets, like this:

```
aMethod(  { puts( "Good morning" ) }  ) # This won't work!
```

Instead we simply put the block right next to the method to which we are passing it, just as we did in the very first example in this chapter. That method receives the block without having to declared a named parameter for it and it calls the block with yield.

Here is a slightly more useful example:

```
def caps( anarg )
  yield( anarg )
end

caps( "a lowercase string" ){ |x| x.capitalize! ; puts( x ) }
```

Here the caps method receives one argument, anarg, and passes this argument to a nameless block which is then executed by yield. When I call the caps method, I pass it a string argument ("a lowercase string") using the normal parameter-passing syntax. The nameless block is passed *after the end* of the parameter list.

When the caps method calls yield(anarg) the string argument, "a lowercase string" is passed into the block; it is assigned to the block variable, x, this capitalizes it and displays it with puts(s).

BLOCKS WITHIN BLOCKS

We've already seen how to use a block to iterate over an array. In the next example, I use one block to iterate over an array of strings, assigning each string in turn to the block variable, s. A second block is then passed to the caps method in order to capitalise the string:

```
["hello","good day","how do you do"].each{
   |s|
   caps( s ){ |x| x.capitalize!
      puts( x )
    }
}
```

This results in this output:

```
Hello
Good day
How do you do
```

PASSING NAMED PROC ARGUMENTS

Up to now, we have passed blocks to procedures either anonymously (in which case the block is executed with the yield keyword) or in the form of a named argument, in which case it is executed using the call method. There is another way to pass a block. When the last argument in a method's list of parameters is preceded by an ampersand (&) it is considered to be a Proc object. This gives you the option of passing an anonymous block to a procedure using the same syntax as when passing a block to an iterator; and yet the procedure itself can receive the block as a named argument. Load **5blocks.rb** to see some examples of this.

5blocks.rb

First, here is a reminder of the two ways which we've already seen of passing blocks. This method has three parameters, a, b, c:

```
def abc( a, b, c )
  a.call
  b.call
  c.call
  yield
end
```

We call this method with three named arguments (which here happen to be blocks but could, in principle, be anything) plus an unnamed block:

```
abc(a, b, c ){ puts "four" }
```

The abc method executes the named block arguments using the call method and the unnamed block using the yield keyword:

```
a.call          #<= call block a
b.call          #<= call block b
c.call          #<= call block c
yield           #<= yield unnamed block: { puts "four" }
```

The next method, abc2, takes a single argument, &d:

```
def abc2( &d )
```

The ampersand here is significant as it indicates that the &d parameter is a block. We don't, however, need to send this block as a named argument. Instead, we pass the unnamed block simply by appending it to the method name:

```
abc2{ puts "four" }
```

Instead of using the yield keyword, the abc2 method is able to execute the block using the name of the argument (without the ampersand):

```
def abc2( &d )
  d.call
end
```

You can think of ampersand-arguments as type-checked block parameters. That is, ampersand arguments are formally declared so unlike nameless blocks (those which are 'yielded') the block doesn't arrive at the method 'unannounced'. But unlike normal arguments (without an ampersand) they must match blocks. You cannot pass some other type of object to abc2:

```
abc2( 10 )  # This won't work!
```

The abc3 method is essentially the same as the abc method apart from the fact that it specifies a fourth formal argument (&d):

```
def abc3( a, b, c, &d)
```

The arguments, a, b and c are called, while the argument &d may be called or yielded, as you prefer:

```
def abc3( a, b, c, &d)
    a.call
    b.call
    c.call
    d.call          #<= block &d
    yield           #<= also block &d
end
```

This means that the calling code must pass to this method three formal arguments plus a block, which may be nameless:

```
abc3(a, b, c){ puts "five" }
```

You can also use a preceding ampersand in order to pass a named block to a method, when the receiving method has no matching named argument, like this:

```
abc3(a, b, c, &myproc )
```

When an ampersanded block variable is passed to a method (as in the code above) it may be yielded. This gives the choice of passing either an unnamed block or a Proc object:

```
xyz{ |a,b,c| puts(a+b+c) }
xyz( &myproc )
```

Be careful, however! Notice in one of the examples above, I have used block parameters (|a,b,c|)with the same names as the three local variables to which I previously assigned Proc objects: a, b, c:

```
a = lambda{ puts "one" }
b = lambda{ puts "two" }
c = proc{ puts "three" }

xyz{ |a,b,c| puts(a+b+c) }
```

Now, in principle, block parameters should be visible only within the block itself. However, it turns out that assignment to block parameters can initialize the values of any local variables with the same name within the block's native scope (see '*What Is A Closure?*' earlier in this chapter).

Even though the variables in the xyz method are named x, y and z, it turns out that the integer assignments in that method are actually made to the variables a, b and c when this block…

```
{ |a,b,c| puts(a+b+c) }
```

… is passed the values of x, y and z:

```
def xyz
  x = 1
  y = 2
  z = 3
  yield( x, y, z )   # 1,2,3 assigned to block parameters a,b,c
end
```

As a consequence, the variables a, b and c within the block's native scope (the main scope of my program) are initialized with the values of the block variables once the code in block has been run:

```
xyz{ |a,b,c| puts(a+b+c) }
puts( a, b, c )                # displays 1, 2, 3
```

To clarify this, try out the simple program in **6blocks.rb**:

6blocks.rb

```
a = "hello world"

def foo
  yield 100
end

puts( a )
foo{ |a| puts( a ) }

puts( a )      #< a is now 100
```

This is an example of one of the pitfalls into which it is all to easy to fall in Ruby. As a general rule, when variables share the same scope (e.g. a block declared within the scope of the main program here), it is best to make their names unique in order to avoid any unforeseen side effects.

Note that the block scoping described here applies to versions of Ruby up to and including Ruby 1.8.x which, at the time of writing, may be considered to be the 'standard' version of Ruby. Changes to scoping are being made in the Ruby 1.9 and will be incorporated in Ruby 2.0. For more on scoping see '*Blocks and Local Variables*' in the Digging Deeper section at the end of this chapter.

PRECEDENCE RULES

Blocks within curly braces have stronger precedence than blocks within do and end. Let's see what that means in practice. Consider these two examples:

```
foo bar do |s| puts( s ) end
foo bar{ |s| puts(s) }
```

Here, foo and bar are methods. So to which method is the block passed? It turns out that the do..end block would be passed to the leftmost method, foo, whereas the block in curly braces would be sent to the rightmost method, bar. This is because curly braces are said to have higher precedence. Consider this program...

precedence.rb

```
def foo( b )
  puts("---in foo---")
  a = 'foo'
  if block_given?
    puts( "(Block passed to foo)" )
     yield( a )
  else
    puts( "(no block passed to foo)" )
  end
  puts( "in foo, arg b = #{b}" )
  return "returned by " << a
end

def bar
  puts("---in bar---")
  a = 'bar'
  if block_given?
    puts( "(Block passed to bar)" )
    yield( a )
  else
    puts( "(no block passed to bar)" )
  end
  return "returned by " << a
end
```

```
foo bar do |s| puts( s ) end        # 1) do..end block

foo bar{ |s| puts(s) }              # 2) {..} block
```

Here the do..end block has lower precedence and the method, foo, is given priority. This means that both bar and the do..end block are passed to foo. Thus, these two expressions are equivalent:

```
foo bar do |s| puts( s ) end
foo( bar ) do |s| puts( s ) end
```

A curly brace block, on the other hand, has stronger precedence so it tries to execute immediately and is passed to the first possible receiver method (bar). The result (that is, the value returned by bar) is then passed as an argument to foo; but this time, foo does not receive the block itself. Thus, the two following expressions are equivalent:

```
foo bar{ |s| puts(s) }
foo( bar{ |s| puts(s) } )
```

If you are confused by all this, take comfort in the fact that you are not alone! The behaviour of Ruby blocks is far from transparent. The potential ambiguities result from the fact that, in Ruby, the parentheses around argument lists are optional. As you can see from the alternative versions I give above, the ambiguities disappear when you use parentheses.

Hint...

A method can test if it has received a block using the block_given? method. You can find examples of this in the **precedence.rb** program.

BLOCKS AS ITERATORS

As mentioned earlier, one of the primary uses of blocks in Ruby is to provide iterators to which a range or list of items can be passed. Many standard classes such as Integer and Array have methods which can supply items over which a block can iterate. For example:

```
3.times{ |i| puts( i ) }

[1,2,3].each{|i| puts(i) }
```

You can, of course, create your own iterator methods to provide a series of values to a block. In the **iterate1.rb** program, I have defined a simple `timesRepeat` method which executes a block a specified number of times. This is similar to the `times` method of the Integer class apart from the fact that it begins at index 1 rather than at index 0 (here the variable `i` is displayed in order to demonstrate this fact):

iterate1.rb

```
def timesRepeat( aNum )
   for i in 1..aNum do
     yield i
  end
end
```

Here is an example of how this method might be called:

```
timesRepeat( 3 ){  |i| puts("[#{i}] hello world") }
```

I've also created a `timesRepeat2` method to iterate over an array:

```
def timesRepeat2( aNum, anArray )
   anArray.each{ |anitem|
     yield( anitem )
  }
end
```

This could be called in this manner:

```
timesRepeat2( 3, ["hello","good day","how do you do"] ){ |x| puts(x) }
```

In fact, of course, it would be better (truer to the spirit of object orientation) if an object itself contained its own iterator method. I've implemented this in the next example. here I have created MyArray, a subclass of Array:

iterate2.rb

```
class MyArray < Array
```

It is initialized with an array when a new MyArray object is created:

```
def initialize( anArray )
  super( anArray )
end
```

It relies upon its own (*self*) `each` method, which is provided by its ancestor, Array, to iterate over the items in the array and it uses the `times` method of Integer to do this a certain number of times. This is the complete class definition:

```
class MyArray < Array
  def initialize( anArray )
    super( anArray )
  end

  def timesRepeat( aNum )
    aNum.times{   # start block 1...
      | num |
      self.each{      # start block 2...
        | anitem |
        yield( "[#{num}] :: '#{anitem}'" )
      }                # ...end block 2
    }               #  ...end block 1
  end
end
```

Notice that, as I have used two iterators (aNum.times and self.each), the times-Repeat method comprises two nested blocks. This is an example of how you might use this...

```
numarr = MyArray.new( [1,2,3] )
numarr.timesRepeat( 2  ){ |x| puts(x) }
```

This would output the following:

```
[0] :: '1'
[0] :: '2'
[0] :: '3'
[1] :: '1'
[1] :: '2'
[1] :: '3'
```

In **iterate3.rb** I have set myself the problem of defining an iterator for an array containing an arbitrary number of sub-arrays, in which each sub-array has the same number of items. In other words it will be like a table or matrix with a fixed number of rows and a fixed number of columns. Here, for example, is a multi-dimensional array with three 'rows' (sub-arrays) and four 'columns' (items):

iterate3.rb

```
multiarr =
[ ['one','two','three','four'],
  [1,   2,   3,     4    ],
  [:a,  :b,  :c,   :d    ]
]
```

I've tried out three alternative versions of this. The first version suffers from the limitation of only working with a pre-defined number (here 2 at indexes [0] and [1]) of 'rows':

```
multiarr[0].length.times{|i|
  puts(multiarr[0][i], multiarr[1][i])
 }
```

The second version gets around this limitation by iterating over each element (or 'row') of `multiarr` and then iteration along each item in that row by obtaining the row length and using the Integer's `times` method with that value:

```
multiarr.each{ |arr|
  multiarr[0].length.times{|i|
    puts(arr[i])
    }
}
```

The third version reverses these operations: the outer block iterates along the length of row 0 and the inner block obtains the item at index `i` in each row:

```
multiarr[0].length.times{|i|
  multiarr.each{ |arr|
    puts(arr[i])
  }
}
```

While versions 2 and 3 work in a similar way, you will find that they iterate through the items in a different order. Run the program to verify that. You could try creating your own subclass of Array and adding iterator methods like this – one method to iterate through the rows in sequence (like Version 2, above) and one to iterate through the columns in sequence (like Version 3).

Digging Deeper

RETURNING BLOCKS FROM METHODS

Earlier, I explained that blocks in Ruby may act as 'closures'. A closure may be said to enclose the 'environment' in which it is declared. Or, to put it another way, it carries the values of local variables from its original scope into a different scope. The example I gave previously showed how the block named ablock captures the value of the local variable x...

block_closure.rb

```
x = "hello world"
ablock = Proc.new { puts( x ) }
```

...and it is then able to 'carry' that variable into a different scope. Here, for example, ablock is passed to aMethod. When ablock is called inside that method it runs the code puts(x). This displays "hello world" and not "goodbye"...

```
def aMethod( aBlockArg )
  x = "goodbye"
  aBlockArg.call                    #<= displays "hello world"
end
```

In this particular example, this behaviour may seem like a curiosity of no great interest. In fact, block/closures can be used more creatively.

For example, instead of creating a block and sending it to a method, you could create a block *inside a method* and return that block to the calling code. If the method in which the block is created happens to take an argument, the block could be initialized with that argument.

This gives us a simple way of creating multiple blocks from the same 'block template', each instance of which is initialized with different data. Here, for example, I have created two blocks, assigned to the variables salesTax and vat, each of which calculates results based on different values (0.10) and (0.175):

block_closure2.rb

```
def calcTax( taxRate )
  return lambda{
    |subtotal|
    subtotal * taxRate
    }
end

salesTax = calcTax( 0.10 )
vat = calcTax( 0.175 )

print( "Tax due on book = ")
print( salesTax.call( 10 ) )          #<= prints: 1.0

print( "\nVat due on DVD = ")
print( vat.call( 10 ) )               #<= prints: 1.75
```

BLOCKS AND INSTANCE VARIABLES

One of the less obvious features of blocks is the way in which they use variables. If a block may truly be regarded as a nameless function or method then, logically, it should be able 1) to contain its own local variables and 2) to have access to the instance variables of the object to which the block belongs.

Let's look first at instance variables. Load up the **closures1.rb** program. This providers another illustration of a block acting as a closure – by capturing the values of the local variables in the scope in which it was created. Here I have created block using the `lambda` method:

closures1.rb

```
aClos = lambda{
  @hello << " yikes!"
}
```

This block appends a string, " yikes!" to an instance variable, `@hello`. Notice that at this stage in the proceedings, no value has previously been assigned to `@hello`.

I have, however, created a separate method, aFunc, which does assign a value to a variable called @hello:

```
def aFunc( aClosure )
  @hello = "hello world"
  aClosure.call
end
```

When I pass my block to this method (the aClosure argument), the aFunc method brings @hello into being. I can now execute the code inside the block using the call method. And sure enough @hello variable contains the string "hello world". The same variable can also be used by calling the block outside of the method. Indeed, now, by repeatedly calling the block, I will end up repeatedly appending the string, " yikes!" to @hello:

```
aFunc(aClos)        #<= @hello = "hello world yikes!"
aClos.call          #<= @hello = "hello world yikes! yikes!"
aClos.call          #<= @hello = "hello world yikes! yikes! yikes!"
aClos.call          # ...and so on
aClos.call
```

If you think about it, this is not really too surprising. After all, @hello is an instance variable so it exists within the scope of an object. When we run a Ruby program, an object called main is automatically created. So we should expect any instance variable created within that object (our program) to be available to everything inside it.

The question, now arises: what would happen if you were to send the block to a method of some *other* object? If that object has its own instance variable, @hello, which variable will the block use – the @hello from the scope in which the block was created or the @hello from the scope of the object in which the block is called? Let's try that out. We'll use the same block as before, except this time it will display a bit of information about the object to which the block belongs and the value of @hello:

```
aClos = lambda{
   @hello << " yikes!"
   puts("in #{self} object of class #{self.class}, @hello = #{@hello}")
}
```

Now to create a new object from a new class (X) and give it a method which will receive our block, b, and call the block:

```
class X
  def y( b )
    @hello = "I say, I say, I say!!!"
    puts( "   [In X.y]" )
    puts("in #{self} object of class #{self.class}, @hello = #{@hello}")
    puts( "   [In X.y] when block is called..." )
    b.call
  end
end

x = X.new
```

To test it out, just pass the block, aClos, to the y method of x:

```
x.y( aClos )
```

And this is what is displayed:

```
  [In X.y]
in #<X:0x32a6e64> object of class X, @hello = I say, I say, I say!!!
   [In X.y] when block is called...
in main object of class Object, @hello = hello world yikes! yikes! yikes!
yikes! yikes! yikes!
```

So, it is clear that the block executes in the scope of the object in which it was *created* (main) and retains the instance variable from that object even though the object in whose scope the block is *called* has an instance variable with the same name and a different value.

BLOCKS AND LOCAL VARIABLES

Now let's see how a block/closure deals with local variables. Load up the **closures2.rb** program. First I declare a variable, x, which is local to the context of the program itself:

closures2.rb

```
x = 3000
```

The first block/closure is called c1. Each time I call this block, it picks up the value of x defined outside the block itself (3000) and returns x + 100:

```
c1 = lambda{
  return x + 100
}
```

This block has no block parameters (that is, there are no 'block local' variables between upright bars) so when it is called with a variable, someval, that variable is discarded, unused. In other words, c1.call(someval) has the same effect as c1.call().

So when you call the block c1, it returns x+100 (i.e. 3100), this value is then assigned to someval. When you call c1 a second time, exactly the same thing happens all over again, so once again someval is assigned 3100:

```
someval=1000
someval=c1.call(someval); puts(someval)        #<= someval is now 3100
someval=c1.call(someval); puts(someval)        #<= someval is now 3100
```

Note: Instead of repeating the call to c1, as show above, you could place the call inside a block and pass this to the times method of Integer like this:

```
2.times{ someval=c1.call(someval); puts(someval) }
```

However, as it can be hard enough to work what just *one* block is up to (such as the c1 block here) I've deliberately avoided using any more blocks than are absolutely necessary in this program!

The second block is named c2. This declares a 'block parameter', z. This too returns a value:

```
c2 = lambda{
  |z|
  return z + 100
}
```

However, this time the returned value can be reused since the block parameter acts like an incoming argument to a method – so when the value of someval is changed after it is assigned the return value of c2 this changed value is subsequently passed as an argument:

```
someval=1000
someval=c2.call(someval); puts(someval)        #<= someval is now 1100
someval=c2.call(someval); puts(someval)        #<= someval is now 1200
```

The third block, c3, looks, at first sight, pretty much the same as the second block, c2. In fact, the only difference is that its block parameter is called x instead of z:

```
c3 = lambda{
  |x|
  return x + 100
}
```

The name of the block parameter has no effect on the return value. As before, someval is first assigned the value 1100 (that is, its original value, 1000, plus the 100 added inside the block) then, when the block is called a second time, someval is assigned the value 1200 (its previous value, 1100, plus 100 assigned inside the block).

But now look at what happens to the value of the local variable x. This was assigned 3000 at the top of the unit. Simply by giving the block parameter the same name, x, we have altered the value of the local variable, x. It now has the value, 1100 – that is, the value that the block parameter, x, last had when the c3 block was called:

```
x = 3000

c3 = lambda{
  |x|
  return x + 100
}

someval=1000
someval=c3.call(someval); puts(someval)
someval=c3.call(someval); puts(someval)
puts( x )                           #<= x is now 1100
```

Incidentally, even though block-local variables and block parameters can affect similarly named local variables outside the block, the block variables themselves have no 'existence' outside the block. You can verify this using the `defined?` keyword to attempt to display the type of variable if it is, indeed, defined:

```
print("x=[#{defined?(x)}],z=[#{defined?(z)}]")
```

Matz, the creator of Ruby, has described the scoping of local variables within a block as 'regrettable'. In particular, he feels that it was mistake to make local variables within a block invisible to the method containing that block. For an example of this, see **local_var_scope.rb:**

local_var_scope.rb

```
def foo
  a = 100
  [1,2,3].each do |b|
    c = b
    a = b
    print("a=#{a}, b=#{b}, c=#{c}\n")
  end
  print("Outside block: a=#{a}\n")       # Can't print #{b} and #{c} here!!!
end
```

Here, the block parameter, `b`, and the block-local variable, `c`, are both visible only when inside the block itself. The block has access to both these variables and to the variable a (local to the foo method). However, outside of the block, b and c are inaccessible and only a is visible.

Just to add to the confusion, whereas the block-local variable, c and the block parameter, b, are both inaccessible outside the block in the example above, they are accessible when you iterate a block with for as in the example below:

```
def foo2
  a = 100
  for b in [1,2,3] do
    c = b
    a = b
    print("a=#{a}, b=#{b}, c=#{c}\n")
  end
  print("Outside block: a=#{a}, b=#{b}, c=#{b}\n")
end
```

In future versions of Ruby, local variables to which values are assigned inside a block (as with c) will also be local to the method (such as foo) outside the block. Formal block parameters, like b, will be local to the block.

CHAPTER ELEVEN

Symbols

Many newcomers to Ruby are confused by symbols. A symbol is an identifier whose first character is a colon (:), so :this is a symbol and so is :that. Symbols are, in fact, not at all complicated – and, in certain circumstances, they may be extremely useful, as we shall see shortly.

Let's first be clear about what a symbol is *not*: it is not a string, it is not a constant and it is not a variable. A symbol is, quite simply, an identifier with no intrinsic meaning other than its own name. Whereas you might assign a value to a variable like this...

```
name = "Fred"
```

...you would *not* assign a value to a symbol. The value of a symbol called :name is :name.

For a more technical account of what a symbol is, refer to the Digging Deeper section at the end of the chapter.

We have, of course, used symbols before. In Chapter 2, for instance, we created attribute readers and writers by passing symbols to the attr_reader and attr_writer methods, like this:

```
attr_reader( :description )
attr_writer( :description )
```

You may recall that the above code causes Ruby to create a @description instance variable plus a pair of getter (reader) and setter (writer) methods called description. Ruby takes the value of a symbol literally. Its value is its name (:description). The attr_reader and attr_writer methods create, from that name, variables and methods with matching names.

SYMBOLS AND STRINGS

It is a common misconception that a symbol is a type of string. After all, isn't the symbol, :hello pretty similar to the string, **"hello"**?

In fact, symbols are quite unlike strings. For one thing, each string is different – so, **"hello"**, **"hello"** and **"hello"** are three separate objects with three separate object_ids.

symbol_ids.rb

```
puts( "hello".object_id ) # These 3 strings have 3 different object_ids
puts( "hello".object_id )
puts( "hello".object_id )
```

But a symbol is unique, so :hello, :hello and :hello all refer to the same object with the same object_id. In this respect, a symbol has more in common with an integer than with a string. Each occurrence of a given integer value, you may recall, refers to the same object so **10**, **10** and **10** may be considered to be the same object and they have the same object_id:

ints_and_symbols.rb

```
# These three symbols have the same object_id
puts( :ten.object_id )
puts( :ten.object_id )
puts( :ten.object_id )

# These three integers have the same object_id
puts( 10.object_id )
puts( 10.object_id )
puts( 10.object_id )
```

Or you could test for equality using the equal? method:

symbols_strings.rb

```
puts( :helloworld.equal?( :helloworld ) )          #=> true
puts( "helloworld".equal?( "helloworld" ) )        #=> false
puts( 1.equal?( 1 ) )                              #=> true
```

Being unique, a symbol provides an unambiguous identifier. You can pass symbols as arguments to methods, like this:

```
amethod( :deletefiles )
```

A method might contain code to test the value of the incoming argument:

symbols_1.rb

```
def amethod( doThis )
  if (doThis == :deletefiles) then
    puts( 'Now deleting files...')
  elsif (doThis == :formatdisk) then
    puts( 'Now formatting disk...')
  else
     puts( "Sorry, command not understood." )
  end
end
```

Symbols could also be used in case statements where they would provide both the readability of strings and the uniqueness of integers:

```
case doThis
  when :deletefiles : puts( 'Now deleting files...')
  when :formatdisk : puts( 'Now formatting disk...')
  else  puts( "Sorry, command not understood." )
end
```

The scope in which a symbol is declared does not affect its uniqueness.

Consider the following...

symbol_ref.rb

```
module One
    class Fred
    end
    $f1 = :Fred
end

module Two
    Fred = 1
    $f2 = :Fred
end

def Fred()
end

$f3 = :Fred
```

Here, the variables `$f1`, `$f2` and `$f3` are assigned the symbol `:Fred` in three different scopes: module One, module Two and the 'main' scope. I'll have more to say on modules in Chapter 12. For now, just think of them as 'namespaces' which define different scopes. And yet each variable refers to the same symbol, `:Fred`, and has the same `object_id`:

```
# All three display the same id!
puts( $f1.object_id )
puts( $f2.object_id )
puts( $f3.object_id )
```

Even so, the 'meaning' of the symbol changes according to its scope.

In other words, in module One, `:Fred` refers to the class `Fred`, in module Two, it refers to the constant, `Fred = 1`, and in the main scope it refers to the method `Fred`.

A rewritten version of the previous program demonstrates this:

symbol_ref2.rb

```
module One
  class Fred
  end
  $f1 = :Fred
  def self.evalFred( aSymbol )
    puts( eval( aSymbol.id2name ) )
  end
end

module Two
  Fred = 1
  $f2 = :Fred
  def self.evalFred( aSymbol )
    puts( eval( aSymbol.id2name ) )
  end
end

def Fred()
  puts( "hello from the Fred method" )
end

$f3 = :Fred

One::evalFred( $f1 )      #=> displays the module::class name: One::Fred
Two::evalFred( $f2 )      #=> displays the Fred constant value: 1
method($f3).call          #=> calls Fred method: displays:
                          #        "hello from the Fred method"
```

Naturally, since the variables `$f1`, `$f2` and `$f3` reference the same symbol, it doesn't matter which variable you use at any given point. The following produces exactly the same results:

```
One::evalFred( $f3 )
Two::evalFred( $f1 )
method($f2).call
```

SYMBOLS AND VARIABLES

symbols_2.rb

To understand the relationship between a symbol and an identifier such as a variable name, take a look at our **symbols_2.rb** program. This begins by assigning the value 1 to a local variable, x. It then assigns the symbol :x to a local variable, xsymbol...

```
x = 1
xsymbol = :x
```

At this point there is no obvious connection between the variable, x, and the symbol :x. I have declared a method which simply takes some incoming argument and inspects and displays it using the p method. I can call this method with the variable and the symbol:

```
# Test 1
amethod( x )
amethod( :x )
```

This is the data which the method prints as a result:

```
1
:x
```

In other words, the value of the x variable is 1, since that's the value assigned to it and the value of :x is :x. But the interesting question that arises is: if the value of :x is :x and this is also the symbolic name of the variable x, would it be possible to use the symbol :x to find the value of the variable x? Confused? Hopefully the next line of code will make this clearer:

```
# Test 2
amethod( eval(:x.id2name))
```

Here, id2name is a method of the Symbol class. It returns the name or string corresponding to the symbol (the to_s method would perform the same function); the end result is that, when given the symbol :x as an argument, id2name returns the string "x". Ruby's eval method (which is defined in the Kernel class) is able to evaluate expressions within strings. In the present case, that means it finds the string "x" and tries to evaluate it as though it were executable code. It finds that x is the name of a variable and that the value of x is 1. So the value 1 is passed to amethod. You can verify this by running **symbols2.rb** and comparing the code with the output.

Evaluating data as code is explained in more detail in Chapter 20.

Things can get even trickier. Remember that the variable, xsymbol has been assigned the symbol :x...

```
x = 1
xsymbol = :x
```

That means that if we eval :xsymbol, we can obtain the name assigned to it – that is, the symbol :x. Having obtained :x we can go on to evaluate this also, giving the value of x – namely, 1:

```
# Test 3
amethod( xsymbol )                                      #=> :x
amethod( :xsymbol )                                     #=> :xsymbol
amethod( eval(:xsymbol.id2name))                        #=> :x
amethod( eval( ( eval(:xsymbol.id2name)).id2name ) )    #=> 1
```

As we've seen, when used to create attribute accessors symbols can refer to method names. We can make use of this by passing a method name as a symbol to the method method (yes, there really is a method called 'method') and then calling the specified method using the call method:

```
#Test 4
method(:amethod).call("")
```

The call method lets us pass arguments and, just for the heck of it, we could pass an argument by evaluating a symbol:

```
method(:amethod).call(eval(:x.id2name))
```

If this seems complicated, take a look at a simpler example in **symbols_3.rb**. This begins with this assignment:

symbols_3.rb

```
def mymethod( somearg )
  print( "I say: " << somearg )
end

this_is_a_method_name = method(:mymethod)
```

Here method(:mymethod) looks for a method with the name specified by the symbol passed as an argument (:mymethod) and, if one is found, it returns the Method object with the corresponding name. In my code I have a method called mymethod and this is now assigned to the variable this_is_a_method_name.

When you run this program, you will see that the first line of output prints the value of the variable:

```
puts( this_is_a_method_name )
            #=> This displays:    #<Method: Object#mymethod>
```

This shows that the variable, this_is_a_method_name, has been assigned the method, mymethod, which is bound to the Object class (as are all methods which are entered as 'freestanding' functions). To double-check that the variable really is an instance of the Method class the next line of code prints out its class:

```
puts( "#{this_is_a_method_name.class}" )
            #=> This displays: Method
```

OK, so if it's really and truly a method, then we should be able to call it, shouldn't we? In order to do that we need to use the call method. That is what the last line of code does:

```
this_is_a_method_name.call( "hello world" )
            #=> This displays: I say: hello world
```

WHY USE SYMBOLS?

Some methods in the Ruby class library specify symbols as arguments. Naturally, if you need to call those methods, you are obliged to pass symbols to them. Other than in those cases, however, there is no absolute requirement to use symbols in your own programming. For many Ruby programmers, the 'conventional' data types such as strings and integers are perfectly sufficient.

Symbols do have a special place in 'dynamic' programming, however. For example, a Ruby program is able to create a new method at runtime by calling, within the scope of a certain class, define_method with a symbol representing the method to be defined and a block representing the code of the method:

add_method.rb

```
class Array
  define_method( :aNewMethod, lambda{ |*args| puts( args.inspect) } )
end
```

After the above code executes, the Array class will have gained a method named aNewMethod. You can verify this by calling method_defined? with a symbol representing the method name:

```
Array.method_defined?( :aNewMethod )       #=> returns: true
```

And, of course, you can call the method itself:

```
[].aNewMethod( 1,2,3 )                      #=> returns: [1,2,3]
```

You can remove an existing method at runtime in a similar way by calling remove_method inside a class with a symbol providing the name of the method to be removed:

```
class Array
  remove_method( :aNewMethod )
end
```

Dynamic programming is invaluable in applications which need to modify the behaviour of the Ruby program itself while that program is still executing. Dynamic programming is widely used in the Rails framework, for example.

Digging Deeper

WHAT IS A SYMBOL?

Previously, I said that a symbol is an identifier whose value is itself. That describes, in a broad sense, the way that symbols behave from the point of view of the Ruby programmer. But it doesn't tell us what symbols are *literally* from the point of view of the Ruby interpreter. A symbol is, in fact, a pointer into the symbol table. The symbol table is Ruby's internal list of known identifiers – such as variable and method names.

If you want to take a peek deep inside Ruby, you can display all the symbols which Ruby knows about like this:

allsymbols.rb

```
p( Symbol.all_symbols )
```

This will shows thousands of symbols including method names such as `:to_s` and `:reverse`, global variables such as `:$/` and `:$DEBUG` and class names such as `:Array` and `:Symbol`. You may restrict the number of symbols displayed using array indexes like this:

```
p( Symbol.all_symbols[0,10] )
```

But you can't sort symbols since symbols are not inherently sequential. The easiest way to display a sorted list of symbols would be to convert them to strings and sort those. In the code below, I pass all the symbols known to Ruby into a block, convert each symbol to a string and collect the strings into a new array which is assigned to the `str_array` variable. Now I can sort this array and display the results:

```
str_arr = Symbol.all_symbols.collect{ |s| s.to_s }
puts( str_arr.sort )
```

CHAPTER TWELVE

Modules and Mixins

In Ruby, each class has only one immediate 'parent', though each parent class may have many 'children'. By restricting class hierarchies to single line of descent, Ruby avoids some of the problems that may occur in those programming languages (such as C++) which permit multiple-lines of descent. When classes have many parents as well as many children; and their parents, and children have yet other parents and children, you risk ending up with an impenetrable network (or 'knotwork'?) rather than the neat, well-ordered hierarchy which you may have intended.

Nevertheless, there are occasions when it is useful for classes which are not closely related to implement some shared features. For example, a Sword might be a type of Weapon but also a type of Treasure; a PC might be a type of Computer but also a type of Investment and so on.

But, since the classes defining Weapons and Treasures or Computers and Investments descend from different ancestor classes, their class hierarchy gives them no obvious way of sharing data and methods. Ruby's solution to this problem is provided by Modules.

A MODULE IS LIKE A CLASS…

The definition of a module looks very similar to a class. In fact, Modules and classes are closely related – the `Module` class is the immediate ancestor of the `Class` class. Just like a class, a module can contain constants, methods and classes. Here's a simple module:

```
module MyModule
  GOODMOOD = "happy"
  BADMOOD = "grumpy"

  def greet
    return "I'm #{GOODMOOD}. How are you?"
  end

end
```

As you can see, this contains a constant, `GOODMOOD` and an 'instance method', `greet`.

MODULE METHODS

In addition to instance methods a module may also have module methods. Just as class methods are prefixed with the name of the class, so module methods are prefixed with the name of the module:

```
def MyModule.greet
  return "I'm #{BADMOOD}. How are you?"
end
```

In spite of their similarities, there are two major features which classes possess but which modules do not: **instances** and **inheritance**. Classes can have instances (objects), superclasses (parents) and subclasses (children); modules can have none of these.

The Module *class* does have a superclass – namely, Object. However, any named modules which you create do *not* have superclasses. For a more detailed account of the relationship between Modules and Classes, see the **Digging Deeper** section at the end of this chapter.

Which leads us to the next question: if you can't create an object from a module, what are modules for? This can be answered in two words: **namespaces** and **mixins**. Ruby's 'mixins' provide a way of dealing with the problem of multiple inheritance. We'll come to mixins shortly. First though, let's look at namespaces.

MODULES AS NAMESPACES

You can think of a module as a sort of named 'wrapper' around a set of methods, constants and classes. The various bits of code inside the module share the same 'namespace' so they are all visible to each other but are not visible to code outside the module.

The Ruby class library defines a number of modules such as Math and Kernel. The Math module contains mathematical methods such as `sqrt` to return a square route and constants such as `PI`. The Kernel module contains many of the methods we've been using from the outset such as `print`, `puts` and `gets`.

Let's assume we have the module which we looked at earlier:

modules1.rb

```
module MyModule
  GOODMOOD = "happy"
  BADMOOD = "grumpy"

  def greet
    return "I'm #{GOODMOOD}. How are you?"
  end

  def MyModule.greet
    return "I'm #{BADMOOD}. How are you?"
  end
end
```

We can access the module constants just as we would access class constants using the :: scope resolution operator like this:

```
puts(MyModule::GOODMOOD)
```

We can access module methods using dot notation – that is, specifying the module name followed by a period and the method name. The following would print out "I'm grumpy. How are you?":

```
puts( MyModule.greet )
```

MODULE 'INSTANCE METHODS'

But how to access the instance method, greet? As the module defines a closed namespace, any code outside the module won't be able to 'see' the greet method so this won't work:

```
puts( greet )
```

If this were a class rather than a module we could, of course, create objects from the class using the new method – and each separate object, each 'instance' of the class - would have access to the instance methods. But you cannot create instances of modules. So how can we use their instance methods? This is where mixins enter the picture…

INCLUDED MODULES OR 'MIXINS'

An object can access the instance methods of a module by including that module using the include method. If you were to include MyModule into your program, everything inside that module would suddenly pop into existence within the current scope. So the greet method of MyModule will now be accessible:

modules2.rb

```
include MyModule
puts( greet )
```

Note that only instance methods are included. In the above example, the greet (instance) method has been included but the MyModule.greet (module) method has not...

```
module MyModule
  GOODMOOD = "happy"
  BADMOOD = "grumpy"

  def greet
    return "I'm #{GOODMOOD}. How are you?"
  end
```

```
    def MyModule.greet
      return "I'm #{BADMOOD}. How are you?"
    end
  end
```

As it's included, the **greet** method can be used just as though it were a normal instance method within the current scope...

```
puts( greet )
```

The process of including a module is also called 'mixing in' – which explains why included modules are often called 'mixins'. When you mix modules into a class definition, any objects created from that class will be able to use the instance methods of the mixed-in module just as though they were defined in the class itself.

modules3.rb

```
class MyClass
  include MyModule

  def sayHi
    puts( greet )
  end

end
```

Not only can the methods of this class access the **greet** method from MyModule, but so too can any objects created from the class:

```
ob = MyClass.new
ob.sayHi
puts(ob.greet)
```

Modules can be thought of as discrete code units which can simplify the creation of reusable code libraries. On the other hand, you might be more interested in using modules as an alternative to multiple inheritance.

Returning to an example which I mentioned right at the start of this chapter, let's assume that you have a Sword class which is not only a type of Weapon but also of Treasure. Maybe Sword is a descendant of the Weapon class (so inherits the Weapon's `deadliness` attribute), but it also needs to have the attributes of a Treasure (such as `value` and `owner`) and, this being an Elvish Sword, of MagicThing. If you define these attributes inside Treasure and MagicThing ***modules*** rather than a Treasure and MagicThing ***classes***, the Sword class would be able to include those modules in order to 'mix in' their methods or attributes:

modules4.rb

```
module MagicThing
  attr_accessor :power
end

module Treasure
  attr_accessor :value
  attr_accessor :owner
end

class Weapon
  attr_accessor :deadliness
end

class Sword < Weapon
 include Treasure
 include MagicThing
 attr_accessor :name
end
```

The Sword object now has access to the methods and attributes of the Sword class itself, of its ancestor class, Weapon, and also of its mixed-in modules, Treasure and MagicThing:

```
s = Sword.new
s.name = "Excalibur"
s.deadliness = "fatal"
s.value = 1000
s.owner = "Gribbit The Dragon"
s.power = "Glows when Orcs Appear"
```

```
puts(s.name)
puts(s.deadliness)
puts(s.value)
puts(s.owner)
puts(s.power)
```

Note, incidentally, that any variables which are local variables in the module cannot be accessed from outside the module. This is the case even if a method inside the module tries to access a local variable and that method is invoked by code from outside the module – for example, when the module is mixed in through inclusion:

mod_vars.rb

```
x = 1                              # local to this program

module Foo
  x = 50                           # local to module Foo

     # This can be mixed in but the variable x won't then be visible
  def no_bar
    return x
  end

  def bar
    @x = 1000
    return  @x
  end
  puts( "In Foo: x = #{x}" )       # this can access the 'module local' x
end

include Foo

puts(x)
puts( no_bar )       # Error! This can't access the module-local variable
                     # needed by the no_bar method
 puts(bar)
```

Note that *instance variables* are available to mixed in methods (such as bar). But *local variables* are not available even when a local variable with the same name exists within the current scope of the mixed in method (so, no_bar is unable to access a variable named x even though x is declared in the current scope).

A module may have its own instance variables which belong exclusively to the module 'object'. These instance variables will be in scope to a module method:

inst_class_vars.rb

```
module X
  @instvar = "X's @instvar"

  def self.aaa
    puts(@instvar)
  end
end

X.aaa #=> "X's @instvar"
```

But instance variables that are referenced in instance objects 'belong' to the scope into which that module is included:

```
module X
  @instvar = "X's @instvar"
    def amethod
      @instvar = 10              # creates @instvar in current scope
      puts(@instvar)
    end
end

include X

X.aaa                    #=> X's @instvar
puts( @instvar )         #=> nil
amethod                  #=> 10
puts( @instvar )         #=> 10
@instvar = "hello world"
puts( @instvar )         #=> "hello world"
```

Class variables are also mixed in and, like instance variables, their values may be reassigned within the current scope:

```
module X
  @@classvar = "X's @@classvar"
end

include X

puts( @@classvar )        #=> X's @classvar
@@classvar = "bye bye"
puts( @@classvar )        #=> "bye bye"
```

You may obtain an array of instance variable names using the `instance_variables` method:

```
p( X.instance_variables )
p( self.instance_variables )
```

NAME CONFLICTS

Module methods (those methods specifically preceded by the module name) can help to protect your code from accidental name conflicts. However, no such protection is given by instance methods within modules. Let's suppose you have two modules – one called Happy and the other called Sad. They each contain a module method called `mood` and an instance method called `expression`.

happy_sad.rb

```
module Happy
  def Happy.mood           # module method
    return "happy"
  end

  def expression           # instance method
    return "smiling"
  end
end
```

```
module Sad
  def Sad.mood              # module method
    return "sad"
  end

  def expression            # instance method
    return "frowning"
  end
end
```

Now, a class, Person, includes both these modules:

```
class Person
  include Happy
  include Sad
  attr_accessor :mood

  def initialize
    @mood = Happy.mood
  end
end
```

The initialize method of the Person class needs to set the value of its @mood variable using the mood method from one if the included modules. The fact that they both have a mood method is no problem; being a module method, mood must be preceded by the module name so Happy.mood won't be confused with Sad.mood.

But both the Happy and Sad modules also contain a method called expression. This is an instance method and, when both the modules are included in the Person class, the expression method can be called without any qualification:

```
p1 = Person.new
puts(p1.expression)
```

Which expression method is object p1 using here? It turns out that it uses the method last defined. In the present case, that happens to be the method defined in the Sad module for the simple reason that Sad is included after Happy. If you change the order of inclusion so that Happy is included after Sad, the `p1` object will use the version of the `expression` method defined in the Happy module.

Before getting carried away with the possibilities of creating big, complex modules and mixing then into your classes on a regular basis, bear this potential problem in mind – i.e. included instance methods with the same name will 'overwrite' one another. The problem may be obvious to spot in my little program. It may not be so obvious in a huge application!

ALIAS METHODS

One way of avoiding ambiguity when you use similarly named methods from multiple modules is to 'alias' those methods. An alias is a copy of an existing method with a new name. You use the `alias` keyword followed by the new name, then the old name:

alias_methods.rb

```
alias  happyexpression expression
```

You can also use `alias` to make copies of methods which have been overridden so that you can specifically refer to a version prior to its overridden definition:

```
module Happy
  def Happy.mood
    return "happy"
  end

  def expression
    return "smiling"
  end
  alias happyexpression expression
end

module Sad
  def Sad.mood
    return "sad"
  end
  def expression
    return "frowning"
  end
  alias sadexpression expression
end

class Person
  include Happy
  include Sad
  attr_accessor :mood
  def initialize
    @mood = Happy.mood
  end
end

p2 = Person.new
puts(p2.mood)                #=> happy
puts(p2.expression)          #=> frowning
puts(p2.happyexpression)     #=> smiling
puts(p2.sadexpression)       #=> frowning
```

MIX-IN WITH CARE!

While each class can only descend from one superclass, it can mix in numerous modules. In fact, it is perfectly permissible to mix one lot of modules into another lot of modules and mix these other modules into classes and those classes into yet more modules. Below, is an example of some code that subclasses classes, mixes in modules and even subclasses classes from within mixed-in modules. This code has been deliberately simplified. For the full horror of a working example, see the sample program, **multimods.rb**:

multimods.rb

```
module MagicThing                 # module
  class MagicClass                # class inside module
  end
end
```

```
module Treasure                      # module
end

module MetalThing
  include MagicThing                 # mixin
  class Attributes < MagicClass      # subclasses class from mixin
  end
end

include MetalThing                   # mixin
class Weapon < MagicClass            # subclass class from mixin
  class WeaponAttributes < Attributes # subclass
  end
end

class Sword < Weapon                 # subclass
 include Treasure                    # mixin
 include MagicThing                  # mixin
end
```

In brief, while modules may, when used with care, help to avoid some of the complexities associated the C++ type of multiple inheritance, they are nonetheless open to misuse. If a programmer really wants to create convoluted hierarchies of classes with inscrutable dependencies on multiple levels of mixed in modules, then he or she can certainly do so. In **multimods.rb** I've shown how easy it is to write a an impenetrable program in just a few lines of code. Imagine what you could do over many thousands of lines of code spread over dozens of code files! Suffice to say, this is not a style of programming which I recommend, so you may want to think carefully before mixing in modules.

INCLUDING MODULES FROM FILES

So far, I've mixed in modules that have been defined within a single source file. Often it is more useful to define modules in separate files and mix them in as needed. The first thing you have to do in order to use code from another file is to load that file using the `require` method, like this:

```
require( "testmod.rb" )
```

Optionally, you may omit the file extension:

```
require( "testmod" )        # this works too
```

The required file must be in the current directory, on the search path or in a folder listed in the predefined array variable `$:`. You can add a directory to this array variable using the usual array-append method, `<<` in this way:

```
$: << "C:/mydir"
```

The global variable, `$:` (a dollar sign and a colon), contains an array of strings representing the directories which Ruby searches when looking for a loaded or required file.

The `require` method returns a `true` value if the specified file is successfully loaded; otherwise it returns `false`. If in doubt, you can simply display the result:

```
puts(require( "testmod.rb" ))
```

Any code which would normally be executed when a file is run will be executed when that file is required. So, if the file, **testmod.rb**, contains this code...

testmod.rb

```
def sing
  puts( "Tra-la-la-la-la....")
end

puts( "module loaded")
sing
```

...and the file, **require_module.rb** contains this...

require_module.rb

```
require( "testmod.rb")
```

...then, when **require_module.rb** is run, this will be the output:

```
module loaded
Tra-la-la-la-la....
```

When a module is declared in the required file, it can be mixed in:

```
require( "testmod.rb")
include MyModule          #mix in MyModule declared in testmod.rb
```

load_module.rb

Ruby also lets you load a file using the load method. In most respects, require and load can be regarded as interchangeable. But there are a few subtle differences. In particular, load can take an optional second parameter which, if this is true, loads and executes the code as an unnamed or anonymous module:

```
load( "testmod.rb", true)
```

The file loaded does not introduce the new namespace into the main program and you will not have access to all the module(s) in the loaded file. When the second argument to load is false or when there is no second argument, you will to have access to modules in the loaded file. Note that you must enter the full file name with load ("testmod" minus the ".rb" extension will not suffice).

Another difference is that require loads a file once only (even if your code requires that file many times) whereas load causes the specified file to be reloaded each time load is called. Let's suppose you have this in the file, **test.rb**:

test.rb

```
MyConst = 1
if @a == nil then
  @a = 1
else
  @a += MyConst
end

puts @a
```

Let's suppose you now require this file three times:

require_again.rb

```
require "test"
require "test"
require "test"
```

This will be the output:

```
1
```

But if you load the file three times...

load_again.rb

```
load "test.rb"
load "test.rb"
load "test.rb"
```

...then this will be the output:

```
1
./test.rb:1: warning: already initialized constant MyConst
2
./test.rb:1: warning: already initialized constant MyConst
3
```

Digging Deeper

MODULES AND CLASSES

We've looked at the behaviour of a module. Let's now find out what a module really *is*. It turns out that, as with most other things in Ruby, a module is an object. Each named module is, in fact, an instance of the Module class:

module_inst.rb

```
module MyMod
end

puts( MyMod.class )              #=> Module
```

You cannot create descendents of *named modules*, so this is not allowed:

```
module MyMod
end

module MyOtherMod < MyMod
end
```

However, as with other classes, it is permissible to create a descendent of the Module *class*:

```
class X < Module
end
```

Indeed, the `Class` class is itself a descendent of the `Module` class. It inherits the behaviour of Module and adds on some important new behaviour – notably the ability to create objects. You can verify that Module is the superclass of Class by running the **modules_classes.rb** program which shows this hierarchy:

modules_classes.rb

```
Class
Module          #=> is the superclass of Class
Object          #=> is the superclass of Module
```

PRE-DEFINED MODULES

The following modules are built into the Ruby interpreter:

> `Comparable, Enumerable, FileTest, GC, Kernel, Math, ObjectSpace, Precision, Process, Signal`

Comparable is a mixin module which permits the including class to implement comparison operators. The including class must define the `<=>` operator, which compares the receiver against another object, returning -1, 0, or +1 depending on whether the receiver is less than, equal to, or greater than the other object. Comparable uses `<=>` to implement the conventional comparison operators (`<`, `<=`, `==`, `>=`, and `>`) and the method `between?`.

Enumerable is a mix-in module for enumeration. The including class must provide the method `each`.

FileTest is a module containing file test functions; its methods can also be accessed from the File class.

The **GC** module provides an interface to Ruby's mark and sweep garbage collection mechanism. Some of the underlying methods are also available via the ObjectSpace module.

Kernel is a module included by the Object class; it defines Ruby's 'built-in' methods.

Math is a module containing module functions for basic trigonometric and transcendental functions. It has both 'instance methods' and module methods of the same definitions and names.

ObjectSpace is a module which contains routines that interact with the garbage collection facility and allow you to traverse all living objects with an iterator.

Precision is a mixin for concrete numeric classes with precision. Here, `precision' means the fineness of approximation of a real number, so, this module should not be included into anything which is not a subset of Real (so it should not be included in classes such as Complex or Matrix).

Process is the module for manipulating processes. All its methods are module methods.

Signal is the module for handling signals sent to running processes. The list of available signal names and their interpretation is system dependent.

Here is a brief overview of three of the most commonly used Ruby modules...

KERNEL

The most important of the pre-defined modules is Kernel which provides many of the 'standard' Ruby methods such as `gets`, `puts`, `print` and `require`. In common with much of the Ruby class library, Kernel is written in the C language. While Kernel is, in fact, 'built into' the Ruby interpreter, conceptually it can be regarded as a mixed-in module which, just like a normal Ruby mixin, makes its methods directly available to any class that requires it. Since it is mixed in to the Object class, from which all other Ruby classes descend, the methods of Kernel are universally accessible.

MATH

math.rb

The Math module's methods are provided as both 'module' and 'instance' methods and can therefore be accessed either by mixing Math into a class or by accessing the module methods 'from the outside' by using the module name, a dot and the method name; you can access constants, using a double-colon:

```
puts( Math.sqrt(144) )
puts( Math::PI )
```

COMPARABLE

compare.rb

The Comparable module provides the neat ability to define your own comparison 'operators' `<`, `<=`, `==`, `>=` and `>` by mixing the module into your class and defining the `<=>` method. You can then specify the criteria for comparing some value from the current object with some other value. You might, for example, compare two integers, the length of two strings or some more eccentric value such as the position of a string in an array. I've opted for this eccentric type of comparison in my example program, `compare.rb`. This uses the index of a string in an array of mythical beings in order to compare the name of one being with that of another. A low index such as 'hobbit' at index 0 is considered to be 'less than' a high index such as 'dragon' at index 6:

```
class Being
  include Comparable
  BEINGS = ['hobbit','dwarf','elf','orc','giant','oliphant','dragon']

  attr_accessor :name

  def <=> (anOtherName)
    BEINGS.index[@name]<=>BEINGS.index[anOtherName]
  end

  def initialize( aName )
    @name = aName
  end

end

elf =  Being.new('elf')
orc = Being.new('orc')
giant = Being.new('giant')

puts( elf.name < orc.name )      #=> true
puts( elf.name > giant.name )    #=> false
```

SCOPE RESOLUTION

As with classes, you may use the double-colon scope resolution operator to access constants (including classes and other modules) declared inside modules. For example, let's suppose you have nested modules and classes, like this:

```
module OuterMod
  moduleInnerMod
    class Class1
    end
  end
end
```

You could use the :: operator to access Class1, like this:

```
OuterMod::InnerMod::Class1
```

See Chapter 2 for an introduction to scope resolution of constants within classes...

Each module and class has its own scope, which means that a single constant name might be used in different scopes. This being so, you could use the :: operator to specify a constant within a precise scope:

```
Scope1::Scope2::Scope3          #...etc
```

If you use this operator at the very start of the constant name this has the effect of 'breaking out' of the current scope and accessing the 'top level' scope:

```
::ACONST                        # refers to ACONST at 'top level' scope
```

The following program provides some examples of the scope operator:

scope_resolution.rb

```
ACONST = "hello"                # We'll call this the 'top-level' constant

module OuterMod
  module InnerMod
    ACONST=10
    class Class1
      class Class2
        module XYZ
          class ABC
            ACONST=100
            def xyz
              puts( ::ACONST ) #<= this prints the 'top-level' constant
            end
          end
        end
      end
    end
  end
end


puts(OuterMod::InnerMod::ACONST)
                    #=> displays 10

puts(OuterMod::InnerMod::Class1::Class2::XYZ::ABC::ACONST)
                    #=> displays 100

ob = OuterMod::InnerMod::Class1::Class2::XYZ::ABC.new
ob.xyz
                    #=> displays hello
```

MODULE FUNCTIONS

module_func.rb

If you want a function to be available both as an instance and a module method, you can use the module_function method with a symbol matching the name of an instance method, like this:

```
module MyModule
  def sayHi
    return "hi!"
  end

  def sayGoodbye
    return "Goodbye"
  end

  module_function :sayHi
end
```

The sayHi method may now be mixed into a class and used as an instance method:

```
class MyClass
  include MyModule
   def speak
      puts(sayHi)
      puts(sayGoodbye)
   end
end
```

It may be used as a module method, using dot notation:

```
ob = MyClass.new
ob.speak
puts(MyModule.sayHi)
```

Since the sayGoodbye method here is not a module function, it cannot be used in this way:

```
puts(MyModule.sayGoodbye)                    #=> Error: undefined
method
```

Ruby uses module_function in some of its standard modules such as Math (in the Ruby library file, **complex.rb**) to create 'matching pairs' of module and instance methods.

EXTENDING OBJECTS

You can add the methods of a module to a specific object (rather than to an entire class) using the extend method, like this:

extend.rb

```
module A
  def method_a
    puts( 'hello from a' )
  end
end

class MyClass
  def mymethod
    puts( 'hello from mymethod of class MyClass' )
  end
end

ob = MyClass.new
ob.mymethod
ob.extend(A)
```

Now that the object ob is extended with the module A, it can access that module's instance method, method_a:

```
ob.method_a
```

You can, in fact, extend an object with several modules all at once. Here, the modules B and C extend the object, ob:

```
ob.extend(B, C)
```

When an object is extended with a module containing a method with the same name as a method in the object's class, the method from the module replaces the method from the class. So, let's assume that ob is extended with this class…

```
module C
  def mymethod
    puts( 'hello from mymethod of module C' )
  end
end
```

Now, when you call ob.mymethod, the string 'hello from mymethod of module C' will be displayed rather than 'hello from mymethod of class MyClass', which was displayed previously.

You can prevent an object from being extended it by 'freezing' it using the freeze method:

```
ob.freeze
```

Any attempt to extend this object further would result in a runtime error. In order to avoid such an error, you can use the frozen? method to test whether or not an object has been frozen:

```
if !(ob.frozen?)
  ob.extend( D )
  ob.method_d
else
  puts( "Can't extend a frozen object" )
end
```

CHAPTER THIRTEEN

Files and IO

Ruby provides classes dedicated to handling IO – Input and Output. Chief among these is a class called, unsurprisingly, IO. The IO class lets you open and close IO 'streams' (sequences of bytes) and read and write data to and from them.

For example, assuming you have a file called 'textfile.txt, containing some lines of text, this is how you might open the file and display each line on screen:

io_test.rb

```
IO.foreach("testfile.txt") {|line| print( line ) }
```

Here `foreach` is a class method of IO so you don't need to create a new IO object in order to use it; instead, you just specify the file name as an argument. The `foreach` method takes a block into which each line that is read from the file is passed as an argument. You don't have to open the file for reading and close it when you've finished (as you might expect from your experience with other languages) as Ruby's `IO.foreach` method does this for you.

IO has a number of other useful methods. For example, you could use the `readlines` method to read the file contents into an array for further processing. Here is a simple example which once again prints the lines to screen:

```
lines = IO.readlines("testfile.txt")
lines.each{|line| print( line )}
```

The File class is a subclass of IO and the above examples could be rewritten using the File class:

file_test.rb

```
File.foreach("testfile.txt") {|line| print( line ) }

lines = File.readlines("testfile.txt")
lines.each{|line| print( line )}
```

OPENING AND CLOSING FILES

While some standard methods open and close files automatically, often, when processing the contents of a file, you will need to open and close the file explicitly. You can open a file using either the `new` or the `open` method. You must pass two arguments to one of those methods – the file name and the file 'mode' – and it returns a new File object. The File modes may be either integers which are defined by operating-system-specific constants or strings. The mode generally indicates whether the file is be opened for reading ("r"), writing ("w") or reading and writing ("rw"). This is the list of available string modes:

Mode	Meaning
"r"	Read-only, starts at beginning of file (default mode).
"r+"	Read-write, starts at beginning of file.
"w"	Write-only, truncates existing file to zero length or creates a new file for writing.
"w+"	Read-write, truncates existing file to zero length or creates a new file for reading and writing.
"a"	Write-only, starts at end of file if file exists, otherwise creates a new file for writing.
"a+"	Read-write, starts at end of file if file exists, otherwise creates a new file for reading and writing.
"b"	(DOS/Windows only) Binary file mode (may appear with any of the key letters listed above).

open_close.rb

Let's look at an actual example of opening, processing and closing files. In **open_close.rb** I first open a file, "myfile.txt", for writing ("w"). When a file is opened for writing, a new file will be created if it doesn't already exist. I use `puts()` to write six strings to the file, one string on each of six lines. Finally I close the file.

```
f = File.new("myfile.txt", "w")
f.puts( "I", "wandered", "lonely", "as", "a", "cloud" )
f.close
```

Closing a file not only releases the 'file handle' (the pointer to the file data) but also 'flushes' any data from memory to ensure that it is all saved onto the file on disk. Failing to close a file may result in unpredictable side-effects (try commenting out the `f.close` shown above to see for yourself!).

Now, having written text into a file, let's see how to open that file and read the data back in. This time I'll read the data in one character at a time. As I do so, I'll keep a count of the characters that have been read. I'll also keep a count of the lines, which will be incremented whenever I read in a linefeed character (given by ASCII code 10). For the sake of clarity, I'll add a string to the end of each line that's been read, displaying its line number. I'll display the characters plus my line-end strings on screen and, when everything has been read from the file I'll close it and display the statistics which I've calculated. Here is the complete code:

```
charcount = 0
linecount = 0
f = File.new("myfile.txt", "r")
while !( f.eof ) do              # while not at end of file...
  c = f.getc()                   # getc gets a single character
  if ( c == 10 ) then            # ...whose ASCII code is tested
    linecount += 1
    puts( " <End Of Line #{linecount}>" )
  else
    putc( c )                    # putc here puts the char to screen
    charcount += 1
  end
end
```

```
if f.eof then
  puts( "<End Of File>" )
end
f.close
puts("This file contains #{linecount} lines and #{charcount} characters." )
```

FILES AND DIRECTORIES…

You can also use the File class to manipulate files and directories on disk. Before attempting to perform some operation on a file, you must naturally make sure that the file exists. It might, after all, have been renamed or deleted after the program started – or the user may have incorrectly entered a file or directory name.

You can verify the existence of a file using the File.exist? method. This is one of several testing methods which are provided to the File class by the FileTest module. As far as the File.exist? method is concerned, a directory counts as a file, so you could use the following code to test for the presence of a C:\ drive (note that you must use double file separator '\\' characters in string, as a single '\' will be treated as an escape character):

file_ops.rb

```
if File.exist?( "C:\\" ) then
  puts( "Yup, you have a C:\\ directory" )
else
  puts( "Eeek! Can't find the C:\\ drive!" )
end
```

If you want to distinguish between a directory and a data file, use the directory? method:

```
def dirOrFile( aName )
  if File.directory?( aName ) then
    puts( "#{aName} is a directory" )
  else
    puts( "#{aName} is a file" )
  end
end
```

COPYING FILES

Let's put the File class to some practical use by writing a simple file backup program. When you run **copy_files.rb** you will be asked to choose a directory to copy from (the source directory) and another directory to copy to (the target directory). Assuming both directories exist, the program will then copy all the files from the source directory to the target directory. If the target directory does not exist, it will ask you if you would like to create it (you should enter "Y" to accept). I've supplied a source directory for you; just enter the name **srcdir** when prompted. When asked for a target directory, enter **targetdir** in order to create a subdirectory of that name beneath the current directory.

The program initializes the variable, sourcedir, with the path of the source directory and it initializes targetdir with the name of the target directory. This is the code that does the file copying:

copy_files.rb

```
Dir.foreach( sourcedir ){
  |f|
  filepath = "#{sourcedir}\\#{f}"
   if !(File.directory?(filepath) ) then
    if File.exist?("#{targetdir}\\#{f}") then
      puts("#{f} already exists in target directory (not copied)" )
    else
      FileUtils.cp( filepath, targetdir )
      puts("Copying... #{filepath}" )
    end
  end
  }
```

Here I've used the foreach method of the Dir class which passes into a block the file name, f, of each file in the specified directory. I'll have more to say about the Dir class shortly. The code constructs a qualified path to the file, filepath, by appending the file name to the directory name given by the sourcedir variable. I only want to copy data files not directories so I test that filepath is a file and not a directory:

```
f !(File.directory?(filepath) )
```

This program won't copy over files that already exist, so it first checks to see if a file with the name, f, already exists in the target directory, targetdir:

```
if File.exist?("#{targetdir}\\#{f}")
```

Finally, assuming all the specified conditions are met, the source file, filepath, is copied to targetdir:

```
FileUtils.cp( filepath, targetdir )
```

Here cp is a file-copy method found in the FileUtils module. This module also contains a number of other useful file handling routines such as mv(source, target) to move a file from source to target; rm(files) to delete one or more files listed in the files parameter and mkdir to create a directory as I have done when creating targetdir in the current program:

```
FileUtils.mkdir( targetdir )
```

DIRECTORY ENQUIRIES

My backup program deals with just one directory level at a time – which is why it tests to see that a file, f, is not a directory before attempting to copy it. There are many times, however, when you may want to traverse the subdirectories. For an example of this, let's write a program that calculates the sizes of all the subdirectories beneath a specified root directory. This might be useful if, for example, you wanted locate the biggest files and directories in order to free up disk space by archiving or deleting them.

Navigating through subdirectories provides us with an interesting programming problem. When we begin searching for the presence of subdirectories we have no idea whether we will find one, none or many. Moreover, any subdirectory we find may contain yet another level of subdirectories each of which may contain other subdirectories and so on through many possible levels.

A DISCURSION INTO RECURSION

Our program needs to be able to navigate down the entire subdirectory tree to any number of levels. In order to be able to do this we have to use recursion.

What Is Recursion?

Put simply, a recursive method is one that calls itself. If you aren't familiar with recursive programming, see 'Recursion Made Simple' in the **Digging Deeper** section at the end of this chapter.

file_info.rb

In the program, **file_info.rb**, the `processfiles` method is recursive:

```
def processfiles( aDir )
  totalbytes = 0
  Dir.foreach( aDir ){
  |f|
  mypath = "#{aDir}\\#{f}"
   s = ""
   if File.directory?(mypath) then
      if f != '.' and f != '..' then
      bytes_in_dir = processfiles(mypath)       # <==== recurse!
      puts( "<DIR> ---> #{mypath} contains [#{bytes_in_dir/1024}] KB" )
   end
   else
     filesize = File.size(mypath)
     totalbytes += filesize
     puts ( "#{mypath} : #{filesize/1024}K" )
   end
 }
  $dirsize += totalbytes
  return totalbytes
end
```

You will see that when the method is first called, down towards the bottom of the source code, it is passed the name of a directory in the variable dirname:

```
processfiles( dirname )
```

I've already assigned the parent of the current directory (given by two dots "..") to dirname. If you are running this program in its original location (that is, its location when extracted from this book's source code archive) this will reference the directory containing subdirectories of all the sample code files. Alternatively, you could assign the name of some directory on your hard disk to the variable, dirname, where indicated in the code. If you do this, don't specify a directory containing huge numbers of files and directories ("**C:\Program Files**" would not be a good choice!) as the program would then take some time to execute.

Let's take a closer look at the code in the processfiles method. Once again, I use Dir.foreach to find all the files in the current directory and pass each file, f, one at a time, to be handled by the code in a block between curly brackets. If f is a directory and is not the current one (".") or its parent directory ("..") then I pass the full path of the directory back to the processfiles method:

```
if File.directory?(mypath) then
   if f != '.' and f != '..' then
      bytes_in_dir = processfiles(mypath)
```

If f is not a directory, but just an ordinary data file, I find its size in bytes with File.size and assign this to the variable, filesize:

```
filesize = File.size(mypath)
```

As each successive file, f, is processed by the block of code, its size is calculated and this value is added to the variable, totalbytes:

```
totalbytes += filesize
```

Once every file in the current directory has been passed into the block, totalbytes will be equal to the total size of all the files in the directory.

However, I need to calculate the bytes in all the subdirectories too. Due to the fact that the method is recursive, this is done automatically. Remember that when the code between curly brackets in the processfiles method determines that the current file, f, is a directory it passes this directory name back to itself – the processfiles method.

Let's imagine that we first call processfiles with the **C:\test** directory. At some point the variable, f, is assigned the name of one of its subdirectories – say **C:\test\dir_a**. Now this subdirectory is passed back to processfiles. No further directories are found in **C:\test\dir_a** so processfiles simply calculates the sizes of all the files in this subdirectory. When it finishes calculating these files, the processfiles method comes to an end and returns the number of bytes in the current directory, totalbytes, to whichever bit of code called the method in the first place:

```
return totalbytes
```

In this case, it was this bit of code inside the processfiles method itself which recursively called the processfiles method:

```
bytes_in_dir = processfiles(mypath)
```

So, when processfiles finishes processing the files in the subdirectory, **C:\test\dir_a**, it returns the total size of all the files found there and this is assigned to the bytes_in_dir variable. The processfiles method now carries on where it left off (that is, it continues from the point at which it called itself to deal with the subdirectory) by processing the files in the original directory, **C:\test**.

No matter how many levels of subdirectories this method encounters, the fact that it calls itself whenever it finds a directory ensures that it automatically travels down every directory pathway it finds, calculating the total bytes in each.

One final thing to note is that the values assigned to variables declared inside the processfiles method will change back to their 'previous' values as each level of recursion completes. So the totalbytes variable will first contain the size of **C:\test\test_a\test_b** then of **C:\test\test_a** and finally of **C:\test**. In order to keep a running total of the combined sizes of all the directories we need to assign values to a variable declared outside of the method. Here I use the global variable, $dirsize for this purpose, adding to it the value of totalbytes calculated for each subdirectory processed:

```
    $dirsize += totalbytes
```

Incidentally, while a byte may be a convenient unit of measurement for very small files, it is generally better to describe larger files in kilobyte sizes and very large files or directories in megabytes. To change bytes to kilobytes or to change kilobytes to megabytes you need to divide by 1024. To change bytes to megabytes, divide by 1048576.

The last line of code in my program does these calculations and displays the results in a formatted string, using Ruby's `printf` method:

```
    printf( "Size of this directory and subdirectories is #{$dirsize} bytes,
            #{$dirsize/1024}K, %0.02fMB",
          "#{$dirsize/1048576.0}" )
```

Notice that I have embedded a formatting placeholder "%0.02fMB" in the first string and have added a second string following a comma:

```
          "#{$dirsize/1048576.0}".
```

The second string calculates the directory size in megabytes and this value is then substituted for the placeholder in the first string. The placeholder's formatting option "%0.02f" ensures that the megabyte value is shown as a floating point number "f", with two decimal places, "0.02".

SORTING BY SIZE

Currently this program prints the file and directory names and their sizes in alphabetical order. But I am more interested in their relative sizes. It would, therefore, be more useful if the files were sorted by size rather than by name.

In order to be able to sort the files we need some way of storing a complete list of all file sizes. One obvious way of doing this would be to add the file sizes to an array. In **file_info2.rb**, I create an empty array, `$files` and, each time a file is processed, I append its size to the array:

file_info2.rb

```
$files << fsize
```

I can then sort the file sizes to display low to high values or (by sorting and then reversing the array), from high to low values:

```
$files.sort                     # sort low to high
$files.sort.reverse             # sort high to low
```

The only trouble with this is that I now end up with an array of file sizes without the associated file names. A better solution would be to use a Hash instead of an Array. I've done this in **file_info3.rb**. First, I create two empty Hashes:

file_info3.rb

```
$dirs = {}
$files = {}
```

Now, when the `processfiles` method encounters a directory, it adds a new entry to the `$dirs` Hash, using the full directory path, `mypath`, as the key and the directory size, `dsize`, as the value:

```
$dirs[mypath] = dsize
```

Key-value pairs are similarly added to the `$files` hash. When the entire structure of subdirectories and files has been processed by recursive calls to the `process-files` method, the `$dirs` hash variable will contain key-value pairs of directory names and sizes and the `$files` hash will contain key-value pairs of file names and sizes.

All that remains now is for these hashes to be sorted and displayed. The standard `sort` method for a Hash sorts the keys, not the values. I want to sort the values (sizes), not the keys (names). In order to do this, I have defined this custom sort method:

```
$files.sort{|a,b| a[1]<=>b[1]}
```

Here the sort method converts the $files Hash into nested arrays of [key,value] pairs and passes two of these, a and b, into the block between curly brackets. The second item (at index [1]) of each [key,value] pair provides the value. The sorting itself is done on the value using Ruby's <=> comparison method. The end result is that this program now displays first a list of files in ascending order (by size) and then a similarly sorted list of directories.

Digging Deeper

RECURSION MADE SIMPLE

recursion.rb

If you've never used recursion before, the recursive 'directory-walking' methods in this chapter may need a little explanation. In order to clarify how recursion works, let's look at a much simpler example. Load up the **recursion.rb** program:

```
$outercount = 0

def addup( aNum )
  aNum += 1
  $outercount +=1
  puts( "aNum is #{aNum}, $outercount is #{$outercount}" )
  if $outercount < 3 then
    addup( aNum ) #<= recursive call to addup method
  end
  puts( "At END: aNum is #{aNum}, outercount is #{$outercount}" )
end

addup( 0 ) #<= This is where it all begins
```

This contains the recursive method, addup, whose sole purpose in life is to count from 1 to 3. The addup method receives an integer value as an incoming argument, aNum.

```
addup( aNum )
```

There is also global variable, $outercount, which lives 'outside' the addup method. Whenever the addup method executes, 1 is added to aNum and 1 is also added to $outercount. Then, just so long as $outercount is less than 3, the code inside the addup method calls the same method (addup) all over again, passing to it the new value of aNum:

```
if $outercount < 3 then
    addup( aNum )
end
```

Let's follow what happens. The whole process is started off by calling addup with the value 0:

```
addup( 0 )
```

The addup method adds 1 to both aNum and $outercount, so both variables now have the value 1. The test ($outercount < 3) evaluates to true, so aNum is passed as an argument to addup. Once again 1 is added to both variables, so aNum is now 2 and $outercount is also 2. Now aNum is once more passed to addup. Yet again 1 is added to both variables giving each the value 3. This time, however, the test condition fails since $outercount is no longer less than 3. So the code that calls addup is skipped and we arrive at the last line in the method:

```
puts( "At END: aNum is #{aNum}, outercount is #{$outercount}" )
```

This prints out the values of aNum and $outercount which, as we expect, are both 3.

Having now arrived at the end of this method, the 'flow of control' moves back to the line of code immediately following the code that originally called the method. Here, the line of code that called the addup method happens to be inside the method itself. Here it is:

```
addup( aNum )
```

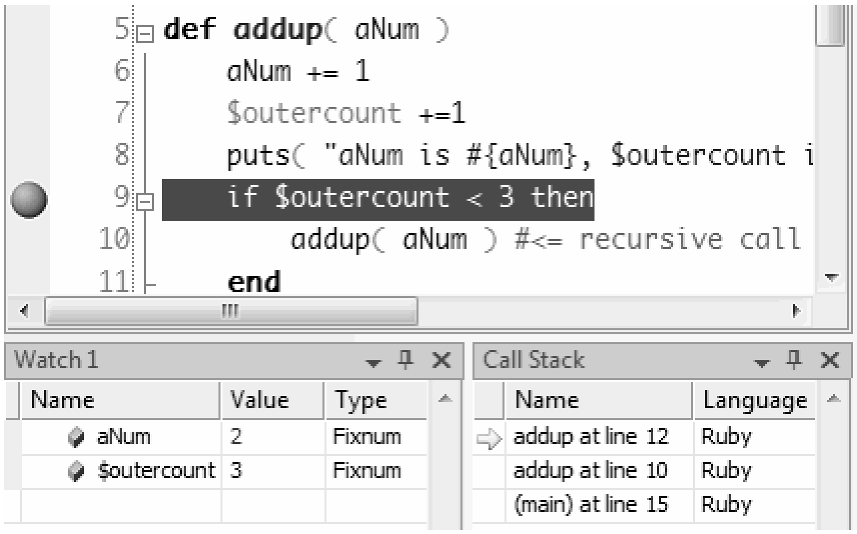
And the first executable line that follows this is (once again) the final line of the method which prints out the values of the two variables:

```
puts( "At END: aNum is #{aNum}, outercount is #{$outercount}" )
```

So we have gone back to an earlier 'point of execution' – the point at which we recursively called the addup method. At that time, the value of aNum was 2 and that is its value now. If this seems confusing, just try to think what would have happened if aNum had been 2 and then we called some other, unrelated, method. On returning from that method, aNum would, of course, still have had the value 2. That's all that's happened here. The only difference is that this method happened to call itself rather than some other method.

Once again the method exits, the control once again returns to the next executable line following the code that called the method – and aNum's value has taken another step back into its own history – it now has the value 1. The $outercount variable, however, lives *outside* the method and is unaffected by recursion so it is still 3.

If you have access to a visual debugger, this entire process will become much clearer if you place a breakpoint on line 9 (if $outercount < 3 then), add aNum and $outercount to the Watch window and repeatedly step into the code after you hit the breakpoint.



```
5   def addup( aNum )
6       aNum += 1
7       $outercount +=1
8       puts( "aNum is #{aNum}, $outercount i
9       if $outercount < 3 then
10          addup( aNum ) #<= recursive call
11      end
```

Watch 1

Name	Value	Type
aNum	2	Fixnum
$outercount	3	Fixnum

Call Stack

Name	Language
addup at line 12	Ruby
addup at line 10	Ruby
(main) at line 15	Ruby

This screenshot shows the recursion program being debugged in Ruby In Steel. I can step through the source code, use the call stack to keep track of the current 'level' of recursion (how many times the addup method has been called) and use the Watch window to monitor the current values of the variables.

CHAPTER FOURTEEN

YAML

At some point, most desktop applications are going to want to save and read structured data to and from disk. We've already seen how to read and write data using simple IO routines such as `gets` and `puts`. But how would you go about writing saving and restoring data from, say, lists of mixed object types? One simple way of doing this with Ruby is by using the YAML.

YAML is an acronym which is (debatably) either short for "Yet Another Markup Language" or (recursively) for "YAML Ain't Markup Language".

CONVERTING TO YAML

YAML defines a serialization (data saving) format which stores information as human-readable text. YAML can be used with a variety of programming languages and, in order to use it in Ruby, your code needs to use the **yaml.rb** file. Generally this would be done by loading or 'requiring' the file at the top of a code unit like this:

```
require 'yaml'
```

Having done this, you will have access to a variety of methods to convert Ruby objects to and from the YAML format so that you can then write their data to a file. Subsequently you will be able to read back this saved data and use it to reconstruct Ruby objects.

To convert an object to YAML format you can use the `to_yaml` method. This will convert any objects – strings, integers, arrays, hashes and so on. For example, this is how you would convert a string:

to_yaml1.rb

```
"hello world".to_yaml
```

And this is how you would convert an array:

```
["a1", "a2" ].to_yaml
```

This is the YAML format which you would obtain as a result of this array conversion:

```
---
- a1
- a2
```

Notice the three dashes which define the start of a new YAML 'document' and the single dashes which define each new element in a list. For more information on the YAML format, refer to the 'Digging Deeper' section at the end of this chapter.

You can also convert objects of non-standard types to YAML. For example, let's suppose you create this class and object…

to_yaml2.rb

```
class MyClass
  def initialize( anInt, aString )
    @myint = anInt
    @mystring =aString
  end
end

ob1 = MyClass.new( 100, "hello world" ).to_yaml
```

The YAML representation of this object will be preceded by the text !ruby/object: followed by the class name, the names of variables with a colon appended (but minus the @) and their values, one per line:

```
--- !ruby/object:MyClass
myint: 100
mystring: hello world
```

If you want to print out the YAML representation of an object, you can use the method y() which is a sort of YAML equivalent of the familiar p() method used to inspect are print normal Ruby objects:

yaml_test1.rb

```
y( ['Bert', 'Fred', 'Mary'] )
```

This displays:

```
---
- Bert
- Fred
- Mary
```

You could similarly could display a Hash...

```
y( { 'fruit' => 'banana', :vegetable => 'cabbage', 'number' => 3 } )
```

...in which case each key/value pair is placed onto a new line:

```
---
number: 3
fruit: banana
:vegetable: cabbage
```

Or you could display your own 'custom' objects...

```
t = Treasure.new( 'magic lamp', 500 )
y( t )
```

...which displays data, as in the earlier example where I used to_yaml, the class name at the top and pairs of variables names and values on successive lines:

```
    --- !ruby/object:Treasure
    name: magic lamp
    value: 500
```

yaml_test2.rb

You can even use y() to display quite complex objects such as nested arrays:

```
arr1 = [   ["The Groovesters", "Groovy Tunes", 12 ],
           [  "Dolly Parton", "Greatest Hits", 38 ]
       ]

y( arr1 )
```

...or arrays containing objects of arbitrary types:

```
arr2 = [   CD.new("The Beasts", "Beastly Tunes", 22),
           CD.new("The Strolling Bones", "Songs For Senior Citizens", 38)
       ]

y( arr2 )
```

NESTED SEQUENCES

When related sequences of data (such as arrays) are nested inside other sequences of data, this relationship is indicated by indentation. So, for example, let's suppose we have this array declared in Ruby...

nested_arrays.rb

```
    arr = [1,[2,3,[4,5,6,[7,8,9,10],"end3"],"end2"],"end1"]
```

When rendered as YAML (e.g. by y(arr)), this becomes:

```
---
- 1
- - 2
  - 3
  - - 4
    - 5
    - 6
    - - 7
      - 8
      - 9
      - 10
    - end3
  - end2
- end1
```

SAVING YAML DATA

Another handy way of turning your Ruby objects into YAML format is provided by the dump method. At its simplest, this converts your Ruby data into YAML format and dumps it into a string:

yaml_dump1.rb

```
arr = ["fred", "bert", "mary"]
yaml_arr = YAML.dump( arr )

    # yaml_arr is now: "--- \n- fred\n- bert\n- mary\n"
```

More usefully, the dump method can take a second argument, which is some kind of IO object, typically a file. You can either open a file and dump data to it…

yaml_dump2.rb

```
f = File.open( 'friends.yml', 'w' )
YAML.dump( ["fred", "bert", "mary"], f )
f.close
```

…or you can open the File (or some other type of IO object) and pass this into an associated block:

```
File.open( 'morefriends.yml', 'w' ){ |friendsfile|
   YAML.dump( ["sally", "agnes", "john" ], friendsfile )
}
```

If you use a block, the file will be closed automatically on exiting the block, otherwise you should explicitly close the file using the `close` method. You can also, incidentally, use a block in a similar way to open a file and read in YAML data:

```
File.open( 'morefriends.yml' ){ |f|
   $arr= YAML.load(f)
}
```

OMITTING VARIABLES ON SAVING

If, for some reason, you want to omit some instance variables when serializing objects, you can do so by defining a method named `to_yaml_properties`.

In the body of this method, place an array of strings. Each string should match the name of the instance variable to be saved. Any variables which are not specified will not be saved. Take a look at this example:

limit_y.rb

```
class Yclass
  def initialize(aNum, aStr, anArray)
    @num = aNum
    @str = aStr
    @arr = anArray
  end

  def to_yaml_properties
    ["@num", "@arr"]      #<= @str will not be saved!
  end
end
```

Here to_yaml_properties limits the variables which will be saved when you call YAML.dump to @num and @arr. The string variable, @str, will not be saved. If you later on wish to reconstruct the objects based on the saved YAML data it is your responsibility to ensure that any 'missing' variables are either not needed (in which case they may be ignored) or, if they are needed, that they are initialized with some meaningful value:

```
ob = Yclass.new( 100, "fred", [1,2,3] )
     # ...creates object with @num=100, @str="fred", @arr=[1,2,3]

yaml_ob = YAML.dump( ob )
     #...dumps to YAML only the @num and @arr data (omits @str)

ob2 = YAML.load( yaml_ob )
     #...creates ob2 from dumped data with @num=100, @arr=[1,2,3]
     # but without @str
```

MULTIPLE DOCUMENTS, ONE FILE

Earlier, I mentioned that three dashes are used to mark the start of a new YAML 'document'. In YAML terms, a document is a discrete group or section. A single file may contain many such 'documents'.

For example, let's assume that you want to save two arrays, arr1, and arr2, to a file, '**multidoc.yml**'. Here arr1 is an array containing two nested arrays and arr2 is an array containing two CD objects:

multi_docs.rb

```
arr1 =  [  ["The Groovesters", "Groovy Tunes", 12 ],
            [  "Dolly Parton", "Greatest Hits", 38 ]
         ]

arr2 = [  CD.new("Gribbit Mcluskey", "Fab Songs", 22),
          CD.new("Wayne Snodgrass", "Singalong-a-Snodgrass", 24) ]
```

This is my routine to dump these arrays to YAML and write them to a file (as explained in Chapter 13, the **'w'** argument causes the file to be opened for writing):

```
File.open( 'multidoc.yml', 'w' ){ |f|
   YAML.dump( arr1, f )
   YAML.dump( arr2, f )
}
```

Look at the file, '**multidoc.yml**' and you'll see that the data has been saved as two separate 'documents' – each one beginning with three dashes:

```
---
- - The Groovesters
  - Groovy Tunes
  - 12
- - Dolly Parton
  - Greatest Hits
  - 38
---
- !ruby/object:CD
  artist: Gribbit Mcluskey
  name: Fab Songs
  numtracks: 22
- !ruby/object:CD
  artist: Wayne Snodgrass
  name: Singalong-a-Snodgrass
  numtracks: 24
```

Now, I need to find a way of reconstructing these arrays by reading in the data as two documents. This is where the `load_documents` method comes to the rescue.

The `load_documents` method calls a block and passes to it each consecutive document. Here is an example of how to use this method in order to reconstruct two arrays (placed inside another array, `$new_arr`) from the two YAML documents:

```
File.open( 'multidoc.yml' ) {|f|
   YAML.load_documents( f ) { |doc|
     $new_arr << doc
   }
}
```

You can verify that `$new_arr` has been initialized with the two arrays by executing the following...

```
puts( "$new_arr contains #{$new_arr.size} elements" )
p( $new_arr[0] )
p( $new_arr[1] )
```

Alternatively, here's a more generic way of doing the same thing, which works with arrays of any length:

```
$new_arr.each{ |arr| p( arr ) }
```

A YAML DATABASE

For an example of a slightly more complicated application which saves and loads data in YAML format, you may want to take a look at **cd_db.rb.** This implements a simple CD database. It defines three types of CD object – a basic CD which contains data on the name, artist and number of tracks and two more specialized descendents – PopCD, which adds data on the genre (e.g. 'rock' or 'country') and ClassicalCD which adds data on the conductor and composer.

When the program is run, the user can enter data to create new CD objects of any of these three types. There is also an option to save data to disk. When the application is run subsequently, the existing data is reloaded.

The data itself is organized very simply (trivially even!) in the code with the data for each object being read into an array before the object itself is created. The whole database of CD objects is saved into a global variable `$cd_arr` and this is written to disk and reloaded into memory using YAML methods:

cd_db.rb

```
def saveDB
   File.open( $fn, 'w' ) {
      |f|
      f.write($cd_arr.to_yaml)
   }
end

def loadDB
   input_data = File.read( $fn )
   $cd_arr = YAML::load( input_data )
end
```

In a real world application, you would, I feel sure, want to create somewhat more elegant data structures to manage your Dolly Parton collection!

ADVENTURES IN YAML

As one final example of using YAML, I've provided an elementary framework for an adventure game (**gamesave_y.rb**). This creates some Treasure objects and some Room objects. The Treasure objects are put 'into' the Room objects (that is, they are placed into arrays contained by the Rooms) and the Room objects are then put into a Map object. This has the effect of constructing a moderately complex data structure in which an object of one type (a Map) contains an arbitrary number of objects of another type (Rooms), each of which may contain zero or more objects of yet other types (Treasures).

At first sight, finding a way of storing this entire network of mixed object types to disk and reconstructing that network at a later stage might look like a programming nightmare.

In fact, thanks to the serialization capabilities supplied by Ruby's YAML library, saving and restoring this data could hardly be easier. This is due to the fact that serialization relieves you of the chore of saving each object one by one. Instead, you only have to 'dump' the top level object – here, that is the Map object, `mymap`.

When this is done, any objects which the top-level object 'contains' (such as Rooms) or which the contained objects themselves contain (such as Treasures) are automatically saved for you. They can then be reconstructed just by loading all the saved data in a single operation and assigning it to the 'top-level' object (here the map):

gamesave_y.rb

```
# Save mymap
File.open( 'game.yml', 'w' ){ |f|
   YAML.dump( mymap, f )
}

# Reload mymap
File.open( 'game.yml' ){ |f|
   mymap = YAML.load(f)
}
```

Digging Deeper

A BRIEF GUIDE TO YAML

In YAML, data is divided up into 'documents' containing 'sequences' of data. Each document begins with three dash characters --- and each individual element in a list begins with a single dash - character. So here, for example, is a YAML data file comprising one document and two list items:

```
---
- artist: The Groovesters
  name: Groovy Tunes
  numtracks: 12
- artist: Dolly Parton
  name: Greatest Hits
  numtracks: 38
```

In the above example, you can see that each list item is made up of two parts – a name such as `artist:` (which is the same in each list item) and a piece of data to its right, such as `Dolly Parton`, which may vary for each list item. These items are like the key-value pairs in a Ruby Hash. YAML refers to key-value lists as 'maps'.

Below is a YAML document containing a list of two items, each of which contains three items – in other words, it is the YAML representation of an array containing two three-item 'nested' arrays:

```
---
- - The Groovesters
  - Groovy Tunes
  - 12
- - Dolly Parton
  - Greatest Hits
  - 38
```

Now let's see how YAML would deal with nested Hashes.

Consider this hash:

hash_to_yaml.rb

```
hsh = { :friend1 => 'mary',
        :friend2 => 'sally',
        :friend3 => 'gary',
        :morefriends => {  :chap_i_met_in_a_bar => 'simon',
                           :girl_next_door => 'wanda'
                         }
}
```

As we've already seen, a Hash is quite naturally represented in YAML as a list of key-value pairs. However, in the example shown above, the key `:morefriends` is associated with a nested hash as its value. How does YAML represent that? It turns out that, as with arrays (see *'Nested Sequences'* earlier in this chapter) it simply indents the nested hash:

```
:friend1: mary
:friend2: sally
:friend3: gary
:morefriends:
    :chap_i_met_in_a_bar: simon
    :girl_next_door: wanda
```

For in-depth information on YAML, see **http://yaml.org**

The YAML libraries supplied with Ruby are quite large and complex and there are many more methods available to you than have been described in this chapter. However, you should now have a enough of an understanding of YAML to use it to good effect in your own programs. You may explore the outer reaches of the YAML libraries at your leisure.

It turns out, though, that YAML is not the only way of serializing data in Ruby. We'll be looking at another way in the next chapter.

CHAPTER FIFTEEN

Marshal

An alternative way of saving and loading data is provided by Ruby's Marshal library. This has a similar set of methods to YAML to enable you to save and load data to and from disk.

SAVING AND LOADING DATA

Compare this program with **yaml_dump2.rb** from the previous chapter:

marshal1.rb

```
f = File.open( 'friends.sav', 'w' )
Marshal.dump( ["fred", "bert", "mary"], f )
f.close

File.open( 'morefriends.sav', 'w' ){ |friendsfile|
   Marshal.dump( ["sally", "agnes", "john" ], friendsfile )
}

File.open( 'morefriends.sav' ){ |f|
   $arr= Marshal.load(f)
}
myfriends = Marshal.load(File.open( 'friends.sav' ))
morefriends = Marshal.load(File.open( 'morefriends.sav' ))
p( myfriends )
p( morefriends )
p( $arr )
```

The two programs are pretty much identical apart from the fact that each occurrence of `YAML` (as in `YAML.dump` and `YAML.load`) has here been replaced with `Marshal`. Moreover, Marshal is 'built in' to Ruby as standard so you don't have to 'require' any extra files in order to use it.

If you look at the data files produced (such as 'friends.sav') you will immediately see that there is a major difference, however. Whereas YAML files are in plain text format, Marshal files are in binary format. So while you may be able to read some characters, such as those in the strings, you won't simply be able to load the saved data and modify it in a text editor.

As with YAML, most data structures can be automatically serialized using Marshal just by dumping the top-level object and loading it when you want to reconstruct all the objects beneath it. For an example, take a look at my little adventure game program. In the last chapter I explained how to save and restore a Map containing Rooms containing Treasures just by dumping and loading the Map object, `mymap` (see **gamesave_y.rb**). The same can be done using Marshal instead of YAML:

gamesave_m.rb

```
File.open( 'game.sav', 'w' ){ |f|
   Marshal.dump( mymap, f )
}

File.open( 'game.sav' ){ |f|
   mymap = Marshal.load(f)
}
```

There are a few special circumstances in which objects cannot be so easily serialized. The code in Ruby's Marshal module (**marshal.c**) documents these exceptions: "If the objects to be dumped include bindings, procedure or method objects, instances of class IO, or singleton objects, a TypeError will be raised". I'll look at an example of this presently when considering how we might go about saving singletons with marshaling.

OMITTING VARIABLES ON SAVING

As with YAML serialization, it is possible to limit the variables that are saved when serializing using Marshal. In YAML we did this by writing a method called to_yaml_properties. With Marshal, we need to write a method named marshal_dump. In the code of this method you should create an array containing the actual variable names to be saved (in YAML, we created an array of strings containing the variable names). This is an example:

limit_m.rb

```
def marshal_dump
  [@num, @arr]
end
```

Another differences is that, with YAML, we were able simply to load the data in order to recreate an object. With Marshal, we need to add a special method called marshal_load to which any data that's been loaded is passed as an argument. This will be invoked automatically when you call Marshal.load and it will be passed the loaded data in the form of an array. The previously saved objects can be parsed from this array. You can also assign values to any variables which were omitted (such as @str here) when the data was saved:

```
def marshal_load(data)
  @num = data[0]
  @arr = data[1]
  @str = "default string"
end
```

Here is a complete program which saves and restores the variables @num and @arr but omits @str:

```
class Mclass
  def initialize(aNum, aStr, anArray)
    @num = aNum
    @str = aStr
    @arr = anArray
  end
```

```
    def marshal_dump
      [@num, @arr]
    end

    def marshal_load(data)
      @num = data[0]
      @arr = data[1]
      @str = "default string"
    end
  end

  ob = Mclass.new( 100, "fred", [1,2,3] )
  p( ob )

  marshal_data = Marshal.dump( ob )
  ob2 = Marshal.load( marshal_data )
  p( ob2 )
```

Note that although the serialization is done here in memory, the same techniques can be used when using Marshal to save and load objects to and from disk.

SAVING SINGLETONS

Let's take a look at a concrete example of a problem mentioned earlier – namely, the inability to use marshaling to save and load a singleton. In **singleton_m.rb** I have create an instance of Object, `ob`, and then extended it in the form of a singleton class with the additional method, `xxx`:

singleton_m.rb

```
  ob = Object.new

  class << ob
    def xxx( aStr )
      @x = aStr
    end
  end
```

The problem arises when I try to save this data to disk using Marshal.dump. Ruby displays an error message stating: “singleton can't be dumped (TypeError)”.

YAML AND SINGLETONS

Before considering how we might deal with this, let’s briefly take a look at how YAML would cope in this situation. The program, **singleton_y.rb** tries to save the singleton shown above using YAML.dump and, unlike Marshal.dump, it succeeds – well, sort of…

singleton_y.rb

```
ob.xxx( "hello world" )

File.open( 'test.yml', 'w' ){ |f|
  YAML.dump( ob, f )
}

ob.xxx( "new string" )

File.open( 'test.yml' ){ |f|
  ob = YAML.load(f)
}
```

If you look at the YAML file which is saved, ‘**test.yml**’, you’ll find that it defines an instance of a plain vanilla Object to which is appended a variable named x which has the string value, “hello world”. That’s all well and good. Except that when you reconstruct the object by loading the saved data, the new ob will be a standard instance of Object which happens to contain an additional instance variable, @x. However, it is no longer the original singleton so the new ob will not have access to any methods (here the xxx method) defined in that singleton. So, while YAML serialization is more permissive about saving and loading data items that were created in a singleton, it does not automatically recreate the singleton itself when the saved data is reloaded.

Let's now return to the Marshal version of this program. The first thing I need to do is find a way of at least making it save and load data items. Once I've done that I'll try to figure out how to reconstruct singletons on reloading.

To save specific data items I can define the marshal_dump and marshal_load methods as explained earlier (see **limit_m.rb**). These should normally be defined in a class from which the singleton derives – *not* in the singleton itself.

This is because, as already explained, when the data is saved, it will be stored as a representation of the class from which the singleton derives. This means that, while you could indeed, add marshal_dump to a singleton derived from class X, when you reconstruct the object, you will be loading data for an object of the generic type X, not of the specific singleton instance.

This code creates a singleton, ob, of class X, saves its data and then recreates a generic object of class X:

singleton_m2.rb

```
class X
  def marshal_dump
    [@x]
  end

  def marshal_load(data)
    @x = data[0]
  end
end

ob = X.new

class << ob
  def xxx( aStr )
    @x = aStr
  end
end
```

```
ob.xxx( "hello" )

File.open( 'test2.sav', 'w' ){ |f|
   Marshal.dump( ob, f )
}

File.open( 'test2.sav' ){ |f|
   ob = Marshal.load(f)
}
```

In terms of the data it contains, the object saved and the object reloaded are identical. However, the object which is reloaded knows nothing about the singleton class and the method `xxx` which this contains forms no part of the reconstructed object. The following, then, would fail:

```
ob.xxx( "this fails" )
```

This Marshal version of the code is, then, equivalent to the YAML version given earlier. It saves and restores the data correctly but it does not reconstruct the singleton.

How, then, is it possible to reconstruct a singleton from saved data? There are, no doubt, many clever and subtle ways in which this might be accomplished. I shall, however, opt for a very simple way:

singleton_m3.rb

```
FILENAME = 'test2.sav'

class X
  def marshal_dump
    [@x]
  end

  def marshal_load(data)
    @x = data[0]
  end
end

ob = X.new
```

```
if File.exists?(FILENAME) then
  File.open(FILENAME){ |f|
    ob = Marshal.load(f)
   }
else
  puts( "Saved data can't be found" )
end

# singleton class
class << ob
  def xxx=( aStr )
    @x = aStr
  end

  def xxx
    return @x
  end
end
```

This code first checks if a file containing the saved data can be found (this sample has been kept deliberately simple - in a real application you would, of course, need to write some exception handling code to deal with the possibility of reading in invalid data). If the file is found, the data is loaded into an object of the generic X type:

```
ob = X.new

if File.exists?(FILENAME) then
  File.open(FILENAME){ |f|
    ob = Marshal.load(f)
 }
```

Only when this has been done is this object 'transformed' into a singleton. Once this is done, the code can use the singleton method xxx on the reconstructed singleton. We can then save the new data back to disk and reload and recreate the modified singleton at a later stage:

```
if ob.xxx == "hello" then
  ob.xxx = "goodbye"
else
  ob.xxx = "hello"
end

File.open( FILENAME, 'w' ){ |f|
   Marshal.dump( ob, f )
}
```

If you wished to save and load singletons in a real application, the singleton 'reconstruction' code could, naturally, be given its own method:

singleton_m4.rb

```
def makeIntoSingleton( someOb )
  class << someOb
    def xxx=( aStr )
      @x = aStr
    end

    def xxx
      return @x
    end
  end
  return someOb
end
```

Digging Deeper

MARSHAL VERSION NUMBERS

The embedded documentation of the Marshal library (a C language file named '**marshal.c**') states the following:

> "Marshaled data has major and minor version numbers stored along with the object information. In normal use, marshaling can only load data written with the same major version number and an equal or lower minor version number."

This clearly raises the potential problem that the format of data files created by marshaling may be incompatible with the current Ruby application. The Marshal version number, incidentally, is not dependent on the Ruby version number so it is not safe to make assumptions of compatibility based solely on the Ruby version.

This possibility of the incompatibility means that we should always check the version number of the saved data before attempting to load it. But how do we get hold of the version number? Once again, the embedded documentation provides a clue. It states:

> "You can extract the version by reading the first two bytes of marshaled data"

And it provides this example:

```
str = Marshal.dump("thing")
RUBY_VERSION        #=> "1.8.0"
str[0]              #=> 4
str[1]              #=> 8
```

OK, so let's try this out in a fully worked piece of code. Here goes…

version_m.rb

```
x = Marshal.dump( "hello world" )
print( "Marshal version: #{x[0]}:#{x[1]}\n" )
```

This prints out:

```
"Marshal version: 4:8"
```

Of course, if you are using a different version of the Marshal library the numbers displayed will be different. In the code above, x is a string and its first two bytes are the major and minor version numbers. The Marshal library also declares two constants, MAJOR_VERSION and MINOR_VERSION, which store the versions number of the Marshal library currently in use. So, at first sight, it looks as though it should be easy to compare the version number of saved data with the current version number.

There is just one problem: when you save data to a file on disk, the dump method takes an IO or File object and it returns an IO (or File) object rather than a string:

version_error.rb

```
f = File.open( 'friends.sav', 'w' )
x = Marshal.dump( ["fred", "bert", "mary"], f )
f.close                        #=> x is now: #<File:friends.sav (closed)>
```

If you now try to get the values of x[0] and x[1], you will receive an error message. Loading the data back from the file is no more instructive:

```
File.open( 'friends.sav' ){ |f|
   x = Marshal.load(f)
}

puts( x[0]  )
puts( x[1]  )
```

The two puts statements here don't (as I was naively hoping) print out the major and minor version numbers of the marshaled data; in fact, they print out the names "fred" and "bert" – that is, the two first items loaded into the array, x, from the data file, 'friends.sav'.

So how the heck can we get the version number from the saved data? I have to admit that I was forced to read may way through the C code (not my favourite activity!) in **marshal.c** and examine the hexadecimal data in a saved file in order to figure this out. It turns out that, just as the documentation says, "You can extract the version by reading the first two bytes of marshaled data". However, this isn't done for you. You have to read this data explicitly – like this:

```
f = File.open('test2.sav')
vMajor = f.getc()
vMinor = f.getc()
f.close
```

Here the `getc` method reads the next 8-bit bye from the input stream. My sample project, **version_m2.rb**, shows a simple way of comparing the version number of the saved data with that of the current Marshal library in order to establish whether the data formats are likely to be compatible before attempting to reload the data.

version_m2.rb

```
if vMajor == Marshal::MAJOR_VERSION then
  puts( "Major version number is compatible" )
  if vMinor == Marshal::MINOR_VERSION then
    puts( "Minor version number is compatible" )
  elsif vMinor < Marshal::MINOR_VERSION then
    puts( "Minor version is lower - old file format" )
  else
    puts( "Minor version is higher - newer file format" )
  end
else
  puts( "Major version number is incompatible" )
end
```

CHAPTER SIXTEEN

Regular Expressions

Regular expressions provide you with powerful ways to find and modify patterns in text – not only short bits of text such as might be entered at a command prompt but also in huge stores of text such as might be found in files on disk.

A regular expression takes the form of a pattern which is compared with a string. Regular expressions also provide means by which you can modify strings so that, for example, you might change specific characters by putting them into uppercase; or you might replace every occurrence of "Diamond" with "Ruby"; or read in a file of programming code, extract all the comments and write out a new documentation file containing all the comments but none of the code. We'll find out how to write a comment-extraction tool shortly. First, though, let's take a look at some very simple regular expressions.

MAKING MATCHES

Just about the simplest regular expression would be a sequence of characters, such as 'abc', which you want to find in a string. A regular expression to match 'abc' can be created simply by placing those letters between two forward-slash delimiters `/abc/`. You can test for a match using the `=~` operator method like this:

regex0.rb

```
puts( /abc/ =~ 'abc' )                  #=> returns 0
```

If a match is made, an integer representing the character position in the string is returned. If no match is made, `nil` is returned.

```
puts( /abc/ =~ 'xyzabcxyzabc' )         #=> returns 3
puts( /abc/ =~ 'xycab' )                #=> returns nil
```

You can also specify a group of characters, between square brackets, in which case a match will be made with any one of those characters in the string. Here, for example, the first match is made with 'c' and that character's position in the string is returned:

```
puts( /[abc]/ =~ 'xycba' )                    #=> returns 2
```

While I've used forward slash delimiters in the examples above, there are alternative ways of defining regular expressions: you can specifically create a new Regexp object initialized with a string or you can precede the regular expression with %r and use custom delimiters - non alphanumeric characters - as you can with strings (*see Chapter 3*). In the example below, I use curly brace delimiters:

regex1.rb

```
regex1 = Regexp.new('^[a-z]*$')
regex2 = /^[a-z]*$/
regex3 = %r{^[a-z]*$}
```

Each of the above, define a regular expression that matches an all-lowercase string (I'll explain the details of the expressions shortly). These expressions can be used to test strings like this:

```
def test( aStr, aRegEx )
  if aRegEx =~ aStr then
    puts( "All lower case" )
  else
    puts( "Not all lower case" )
  end
end

test( "hello", regex1 )    #=> matches: "All lower case"
test( "hello", regex2 )    #=> matches: "All lower case"
test( "Hello", regex3 )    #=> no match: "Not all lower case"
```

To test for a match you can use `if` and the `=~` operator:

if_test.rb

```
if /def/ =~ 'abcdef'
```

The above expression evaluates to true if a match is made (and an integer is returned); it would evaluate to false if no match were made (and `nil` were returned):

```
RegEx = /def/
Str1  = 'abcdef'
Str2  = 'ghijkl'

if RegEx =~ Str1 then
  puts( 'true' )
else
  puts( 'false' )
end                          #=> displays: true

if RegEx =~ Str2 then
  puts( 'true' )
else
  puts( 'false' )
end                          #=> displays: false
```

Frequently, it is useful to attempt to match some expression from the very start of a string; the character `^` followed by a match term is used to specify this. It may also be useful to make a match from the end of the string; the character `$` preceded by a match term is used to specify that.

start_end1.rb

```
puts( /^a/ =~ 'abc' )          #=> returns 0
puts( /^b/ =~ 'abc' )          #=> returns nil
puts( /c$/ =~ 'abc' )          #=> returns 2
puts( /b$/ =~ 'abc' )          #=> returns nil
```

Matching from the start or end of a string becomes more useful when it forms a part of a more complex pattern. Often such a pattern tries to match zero or more instances of a specified pattern. The * character is used to indicate zero or more matches of the pattern which it follows. Formally, this is known as a 'quantifier'. Consider this example:

start_end2.rb

```
puts( /^[a-z 0-9]*$/ =~ 'well hello 123' )
```

Here, the regular expression specifies a range of characters between square brackets. This range includes all lowercase characters, a-z, all digits, 0-9, plus the space character (that's the space between the 'z' and the '0' in this expression). The ^ character means that the match must be made from the start of the string, the * after the range means that zero or more matches with the characters in the range must be made and the $ character means that the matches must be made right up to the end of the string. In other words, this pattern will only match a string containing lowercase characters, digits and spaces from the start right to the end of the string:

```
puts( /^[a-z 0-9]*$/ =~ 'well hello 123' )
                                    # match at 0
puts( /^[a-z 0-9]*$/ =~ 'Well hello 123' )
                                    # no match due to ^ and uppercase 'W'
```

Actually, this pattern will also match an empty string, since * indicates that *zero or more* matches are acceptable:

```
puts( /^[a-z 0-9]*$/ =~ '' )            # this matches!
```

If you want to exclude empty strings, use + (to match *one or more* occurrences of the pattern):

```
puts( /^[a-z 0-9]+$/ =~ '' )            # no match
```

Try out the code in **start_end2.rb** for more examples of ways in which ^, $, * and + may be combined with ranges to create a variety of different match-patterns.

You could use these techniques to determine specific characteristics of strings, such as whether or not a given string is uppercase, lowercase or mixed case:

regex2.rb

```
aStr = "HELLO WORLD"

case aStr
  when /^[a-z 0-9]*$/
    puts( "Lower case" )
  when /^[A-Z 0-9]*$/
    puts( "Upper case" )
  else
    puts( "Mixed case\n" )
end
```

Often regular expressions are used to process the text in a file on disk. Let's suppose, for example, that you want to display all the full-line comments in a Ruby file but omit all the code or partial line-comments. You could do this by trying to match from the start of each line (`^`) zero or more whitespace characters (a whitespace character is represented by `\s`) up to a comment character ('`#`'):

regex3a.rb

```
# displays all the full-line comments in a Ruby file
File.foreach( 'regex1.rb' ){ |line|
  if line =~ /^\s*#/ then
    puts( line )
  end
}
```

MATCH GROUPS

You can also use a regular expression to match one or more substrings. In order to do this, you should put part of the regular expression between round brackets. Here I have two groups (sometimes called 'captures'), the first tries to match the string 'hi', the second tries to match a string starting with 'h', followed by any three characters (a dot means 'match any single character' so the three dots here will match any three consecutive characters) and ending with 'o':

groups.rb

```
/(hi).*(h...o)/ =~ "The word 'hi' is short for 'hello'."
```

After evaluating groups in a regular expression, a number of variables, equal to the number of groups, will be assigned the matched value of those groups. These variables take the form of a $ followed by a number: $1, $2, $3 and so on. After executing the above code, I can access the variables, $1 and $2 like this:

```
print( $1, " ", $2, "\n" )          #=> displays: hi hello
```

Note that if the entire regular expression is unmatched, none of the group variables with be initialized. If, for example, 'hi' is in the string but 'hello' isn't, neither of the group variables will be initialized. Both would be nil.

Here is another example, which returns three groups, each of which contains a single character (given by the dot). Groups $1 and $3 are then displayed:

```
/(.)(.)(.)/ =~ "abcdef"
print( $1, " ", $3, "\n" )          #=> displays: a c
```

Here is a new version of the comment-matching program which was given earlier (**regex3a.rb**); this has now been adapted to use the value of the group (*.) which returns all the characters (zero or more) following the string matched by the preceding part of the regular expression (which here is: ^\s*#). This matches zero or more whitespace (\s*) characters from the start of the current line (^) up to the first occurrence of a hash or pound character: #:

regex3b.rb

```
File.foreach( 'regex1.rb' ){ |line|
  if line =~ /^\s*#(.*)/ then
    puts( $1 )
  end
}
```

The end result of this is that only lines in which the first printable character is # are matched; and $1 prints out the text of those lines minus the # character itself. As we shall see shortly, this simple technique provides the basis for a useful tool for extracting documentation from a Ruby file.

You aren't limited merely to extracting and displaying characters verbatim; you can also modify text. This example, displays the text from a Ruby file but changes all Ruby line-comment characters ('#') preceding full-line comments to C-style line-comments ('//'):

regex4.rb

```
File.foreach( 'regex1.rb' ){ |line|
  line = line.sub(/(^\s*)#(.*)/, '\1//\2')
    puts( line )
}
```

In this example, the `sub` method of the String class has been used; this takes a regular expression as its first argument (here `/(^\s*)#(.*)/`) and a replacement string as the second argument (here `'\1//\2'`) . The replacement string may contain numbered place-holders such as `\1` and `\2` to match any groups in the regular expression - here there are two groups between round brackets: `(^\s*)` and `(.*)`. The `sub` method returns a new string in which the matches made by the regular expression are substituted into the replacement string, while any unmatched elements (here the `#` character, are omitted).

MATCHDATA

The `=~` 'operator' is not the only means of finding a match. The Regexp class also has a `match` method. This works in similar way to `=~` but, when a match is made, it returns a MatchData object rather than an integer. A MatchData object contains the result of a pattern match. At first sight, this may appear to be a string…

match.rb

```
puts( /cde/ =~ 'abcdefg' )              #=> 2
puts( /cde/.match('abcdefg') )          #=> cde
```

In fact, it is an instance of the MatchData class which contains a string:

```
p( /cde/.match('abcdefg') )             #=> #<MatchData:0x28cedc8>
```

A MatchData object may contain groups or 'captures' and these can be returned in an array using either the `to_a` or `captures` method, like this:

matchdata.rb

```
x = /(^.*)(#)(.*)/.match( 'def myMethod # This is a very nice method' )
x.captures.each{ |item| puts( item ) }
```

The above displays:

```
def myMethod
#
 This is a very nice method
```

Note that there is a subtle difference between the `captures` and the `to_a` methods. The first returns only the captures:

```
x.captures          #=>["def myMethod ","#"," This is a very nice method"]
```

The second returns the original string (at index 0) followed by the captures:

```
x.to_a              #=>["def myMethod # This is a very nice method","def
                    myMethod ","#"," This is a very nice method"]
```

PRE AND POST MATCH

The MatchData class supplies the `pre_match` and `post_match` methods to return the strings preceding or following a match. Here, for example, we make a match on the comment character, '`#`':

pre_post_match.rb

```
x = /#/.match( 'def myMethod # This is a very nice method' )
puts( x.pre_match )       #=> def myMethod
puts( x.post_match )      #=>  This is a very nice method
```

Alternatively, you can use the special variables, `$`` (with a backquote) and `$'` (with a normal quote) to access pre and post matches respectively:

```
x = /#/.match( 'def myMethod # This is a very nice method' )
puts( $` )                  #=> def myMethod
puts( $' )                  #=>  This is a very nice method
```

When using `match` with groups, you can use array-style indexing to obtain specific items. Index 0 is the original string; higher indexes are the groups:

match_groups.rb

```
puts( /(.)(.)(.)/.match("abc")[2] )         #=> "b"
```

The special variable, `$~` can be used to access the last MatchData object and once again you can refer to groups using array-style indexing:

```
puts( $~[0], $~[1], $~[3] )
```

However, in order to use the full range of methods of the Array class, you must use `to_a` or `captures` to return the match groups as an array:

```
puts( $~.sort )             # this doesn't work!
puts( $~.captures.sort )  # this does
```

GREEDY MATCHING

When a string contains more than one potential match, you may sometimes want to return the string up to the first match (that is, as little of the string as possible consistent with the match pattern) and at other times you may want the string up to the last match (that is, as much of the string as possible).

In the latter case (getting as much of the string as possible) the match is said to be 'greedy'. The `*` and `+` pattern quantifiers are greedy. You can put them on a diet, to make them return the least possible, by putting `?` after them:

greedy1.rb

```
puts( /.*at/.match('The cat sat on the mat!') )
     #=> returns: The cat sat on the mat
puts( /.*?at/.match('The cat sat on the mat!') )
     #=> returns: The cat
```

You can control the greediness of pattern matching to do things such as process directory paths:

greedy2.rb

```
puts( /.+\\/.match('C:\mydirectory\myfolder\myfile.txt') )
     #=> C:\mydirectory\myfolder\
puts( /.+?\\/.match('C:\mydirectory\myfolder\myfile.txt') )
     #=> C:\
```

STRING METHODS

Up to now, we've used methods of the Regexp class when processing strings. In fact, pattern matching can go both ways due to the fact that the String class has a few regular expression methods of its own. These include `=~` and `match` (so you can switch the order of the String and Regexp objects when matching), plus the `scan` method which iterates through a string looking for as many matches as possible. Each match is added to an array. Here, for example, I am looking for matches on the letters 'a', 'b' or 'c'. The `match` method returns the first match ('a') wrapped up in a MatchData object; but the `scan` method keeps scanning along the string and returns all the matches it finds as elements in an array:

match_scan.rb

```
TESTSTR = "abc is not cba"
puts( "\n--match--" )
b = /[abc]/.match( TESTSTR )
     #=> MatchData: "a"
puts( "--scan--" )
a = TESTSTR.scan(/[abc]/)
     #=> Array: ["a", "b", "c", "c", "b", "a"]
```

The scan method may optionally be passed a block so that the elements of the array created by scan can be processed in some way:

```
a = TESTSTR.scan(/[abc]/){|c| print( c.upcase ) }
    #=> ABCCBA
```

A number of other String methods can be used with regular expressions. One version of the String.slice method takes a regular expression as an argument and returns any matched substring. The String.slice! method (note the ! at the end) deletes the matched substring from the receiver string and returns the substring:

string_slice.rb

```
s = "def myMethod # a comment "

puts( s.slice( /m.*d/ ) )   #=> myMethod
puts( s )                   #=> def myMethod # a comment
puts( s.slice!( /m.*d/ ) )  #=> myMethod
puts( s )                   #=> def  # a comment
```

The split method splits a string into substrings, based on a pattern. The results (minus the pattern) are returned as an array; an empty pattern splits a string into characters:

string_ops.rb

```
s = "def myMethod # a comment"

p( s.split( /m.*d/ ) )
    # =>   ["def ", " # a comment"]
p( s.split( /\s/ ) )
    #=>    ["def", "myMethod", "#", "a", "comment"]
p( s.split( // ) )
    # =>   ["d", "e", "f", " ", "m", "y", "M", "e", "t", "h", "o", "d", " ",
           "#", " ", "a", " ", "c", "o", "m", "m", "e", "n", "t"]
```

You can use the sub method to match a regular expression and replace its first occurrence with a string. If no match is made, the string is returned unchanged:

```
s = "def myMethod # a comment"
s2 = "The cat sat on the mat"
p( s.sub( /m.*d/, "yourFunction" ) ) #=> "def yourFunction # a comment"
p( s2.sub( /at/, "aterpillar" ) )       #=> "The caterpillar sat on the mat"
```

The sub! method works like sub but modifies the original (receiver) string. Alternatively you can use the gsub method (or gsub! to modify the receiver) to substitute all occurrences of the pattern with a string:

```
p( s2.gsub( /at/, "aterpillar" ) )
     #=> "The caterpillar saterpillar on the materpillar"
```

FILE OPERATIONS

I said earlier that regular expressions are often used to process data stored in files on disk. In some earlier examples, we read in data from a disk file, did some pattern-matching and displayed the results on screen. Here is one more example in which we count the words in a file. This is done by scanning each line in order to create an array of words (that is, sequences of alphanumeric characters) then adding the size of each array to the variable, count:

wordcount.rb

```
count = 0
File.foreach( 'regex1.rb' ){ |line|
  count += line.scan( /[a-z0-9A-Z]+/ ).size
}
puts( "There are #{count} words in this file." )
```

I've include some alternative code (commented) in the sample program which displays each word along with its number:

```
File.foreach( 'regex1.rb' ){ |line|
  line.scan( /[a-z0-9A-Z]+/ ).each{ |word|
    count +=1
    print( "[#{count}] #{word}\n" )
  }
}
```

Now let's see how to deal with two files at once – one for reading, another for writing. This first example opens the file **testfile1.txt** for writing and passes the file variable, f, into a block. I now open a second file, **regex1.rb**, for reading and use File.foreach to pass into a second block each line of text read from this file. I use a simple regular expression to create a new string to match lines with Ruby-style comments; the code substitutes C-style comment characters ('//') for the Ruby comment character ('#') when that character is the first non-whitespace character on a line; and writes each line to **testfile1.txt** with code-lines unmodified (as there are no matches on those) and comment-lines changed to C-style:

regexp_file1.rb

```
File.open( 'testfile1.txt', 'w' ){ |f|
  File.foreach( 'regex1.rb' ){ |line|
    f.puts( line.sub(/(^\s*)#(.*)/, '\1//\2')  )
  }
}
```

This illustrates just how much can be done with regular expressions and very little coding.

This next example show how you might read in one file (here the file, **regex1.rb**) and write out two new files – one of which (**comments.txt**) contains only line comments, while the other (**nocomments.txt**) contains all the other lines:

regexp_file2.rb

```
file_out1 = File.open( 'comments.txt', 'w' )
file_out2 = File.open( 'nocomments.txt', 'w' )

File.foreach( 'regex1.rb' ){ |line|
  if line =~ /^\s*#/ then
    file_out1.puts( line )
  else
    file_out2.puts( line )
  end
}

file_out1.close
file_out2.close
```

Digging Deeper

REGULAR EXPRESSIONS

This is a list of some of the elements which can be used in regular expressions...

^	beginning of a line or string
$	end of a line or string
.	any character except newline
*	0 or more previous regular expression
*?	0 or more previous regular expression (non greedy)
+	1 or more previous regular expression
+?	1 or more previous regular expression (non greedy)
[]	range specification (e.g. [a-z] means a character in the range 'a' to 'z')
\w	an alphanumeric character
\W	a non-alphanumeric character
\s	a whitespace character
\S	a non-whitespace character
\d	a digit
\D	a non-digit character
\b	a backspace (when in a range specification)
\b	word boundary (when not in a range specification)
\B	non-word boundary
*	zero or more repetitions of the preceding
+	one or more repetitions of the preceding
{m,n}	at least m and at most n repetitions of the preceding
?	at most one repetition of the preceding
|	either the preceding or next expression may match
()	a group

Here are a few more sample regular expressions...

overview.rb

```
# match chars...
puts( 'abcdefgh'.match( /cdefg/ ) )     # literal chars
            #=> cdefg
puts( 'abcdefgh'.match( /cd..g/ ) )     # dot matches any char
            #=> cdefg

# list of chars in square brackets...
puts( 'cat'.match( /[fc]at/ )
            #=> cat
puts( "batman's father's cat".match( /[fc]at/ ) )
            #=> fat
puts( 'bat'.match( /[fc]at/ ) )
            #=> nil

# match char in a range...
puts( 'ABC100x3Z'.match( /[A-Z][0-9][A-Z0-9]/ ) )
            #=> C10
puts( 'ABC100x3Z'.match( /[a-z][0-9][A-Z0-9]/ ) )
            #=> x3Z

# escape 'special' chars with \
puts( 'ask who?/what?'.match( /who\?\/w..t\?/ ) )
            #=> who?/what?
puts( 'ABC 100x3Z'.match( /\s\S\d\d\D/ ) )
            #=>  " 100x" (note the leading space)

# scan for all occurrences of pattern 'abc' with at least 2 and
# no more than 3 occurrences of the letter 'c'
p( 'abcabccabcccabccccabccccccabccccccccc'.scan( /abc{2,3}/ ) )
            #=> ["abcc", "abccc", "abccc", "abccc", "abccc"]

# match either of two patterns
puts( 'my cat and my dog'.match( /cat|dog/ ) )        #=> cat
puts( 'my hamster and my dog'.match( /cat|dog/ ) )   #=> dog
```

CHAPTER SEVENTEEN

Threads

There may be times when your programs need to perform more than one action at a time. For example, maybe you want to do some disk operations and simultaneously display some feedback to the user. Or you might want to copy or upload some files 'in the background' while still allowing the user to carry on with some other task 'in the foreground'.

In Ruby, if you want to do more than one task at a time, you can run each task in its own 'thread'. A thread is like a program within a program. It runs some particular piece of code independently of any other threads.

However, as we shall see shortly, multiple threads may need to find ways of cooperating with each other so that, for example, they can share the same data and they don't hog all the processing time available to themselves, thereby preventing other threads from running.

CREATING THREADS

Threads can be created like any other object, using the `new` method. When you do this you must pass to the Thread a block containing the code which you want the thread to run.

What follows is my first attempt at creating two threads, one of which should print four strings while the other prints ten numbers:

threads1.rb

```
# This is a simple threading example which, however,
# doesn't work as anticipated!

words = ["hello", "world", "goodbye", "mars" ]
numbers = [1,2,3,4,5,6,7,8,9,10]

Thread.new{
    words.each{ |word| puts( word ) }
}

Thread.new{
    numbers.each{ |number| puts( number ) }
}
```

In all probability, when you run this you may see nothing or, anyway, very little. I've added a report of the time taken for the program to execute and this shows that the darn' thing finishes before it has time to get started!

RUNNING THREADS

Here is a simple fix to the thread-running problem. Right at the end of the code, add this:

threads2.rb

```
sleep( 5 )
```

Now when you run the code again, you should see all the strings and all the numbers, albeit a bit jumbled up. This is, in fact, exactly what we want since it shows that time is now being divided up between the two threads; thats' why the words and numbers are jumbled – first one thread executes and displays a word, then the next thread executes and displays a number, then execution returns to the first thread and so on until the first thread ends (when all four words have been displayed) at which point the second thread can run without interruption.

Now compare this with the first version of the program. In that program I created two threads but, just as Ruby was just getting itself ready to run the code they contained, *bam!*, it arrived at the end of the program and shut everything down – including my two threads. So, in effect, the threads were killed off before they had time to do anything.

By adding a call to `sleep( 5 )` I've given Ruby a five second delay – plenty of time to run those threads before the program exits. There is just one problem with this technique – and it's a big problem. Adding unnecessary delays to your programs in order to let threads run defeats the object of the exercise. The timer display here shows that the program takes all of five whole seconds to run – which is about 4.99 seconds or so longer than is strictly necessary! We'll be looking at more civilized ways of handling threads shortly.

Going Native?

Currently (in Ruby 1.8x) Ruby's threads are not 'native'. Put simply, this means that Ruby threads exist inside the closed world of a Ruby program - multiple threads each being allocated time (using a procedure called 'time-slicing') within a single process. Ruby does not take advantage of 'native threads' which are handled by the operating system to allow more efficient execution (using 'pre-emptive multitasking') on one or more processors. While Ruby threads sacrifice efficiency, they do at least benefit from portability; threads written on one operating system will also run on a different operating system. Subsequent versions of Ruby (at the time of writing Ruby 1.9 may be regarded as an 'experimental' version leading towards Ruby 2.0) will support native threading.

THE MAIN THREAD

Even if you don't explicitly create any threads, there is always at least one thread executing – the main thread in which your Ruby program is running. You can verify this by entering the following:

```
p( Thread.main )
```

This will display something like this:

```
#<Thread:0x28955c8 run>
```

Here, Thread it the thread's class, 0x28955c8 (or some other number) is its hexadecimal object identifier and run is the thread's current status.

THREAD STATUS

Each thread has a status which may be one of the following:

run	when thread is executing
sleep	when thread is sleeping or waiting on I/O
aborting	when thread is aborting
false	when thread terminated normally
nil	when thread terminated with an exception

You can obtain the status of a thread using the status method. The status is also shown when you inspect a thread, in which case either a nil or a false status is shown as 'dead'.

thread_status.rb

```
puts( Thread.main.inspect )                #=> #<Thread:0x28955c8 run>
puts( Thread.new{ sleep }.kill.inspect )   #=> #<Thread:0x28cddc0 dead>
puts( Thread.new{ sleep }.inspect )        #=> #<Thread:0x28cdd48 sleep>
thread1 = Thread.new{ }
puts( thread1.status )                     #=> false
thread2 = Thread.new{ raise( "Exception raised!" ) }
puts( thread2 )                            #=> nil
```

ENSURING THAT A THREAD EXECUTES

Let's return to the problem we had in our previous programs. Recall that we created two threads but the program finished before either of them had a change to run. We fixed this by inserting a fixed length delay using the sleep method.

But deliberately introducing gratuitous delays into your programs is not something you would want to do as a general rule. Fortunately, Ruby has a more civilized way of ensuring that a thread has time to execute. The join method forces the calling thread (for example the main thread) to suspend its own execution (so it doesn't just terminate the program) until the thread which calls join has completed:

join.rb

```
words = ["hello", "world", "goodbye", "mars" ]
numbers = [1,2,3,4,5,6,7,8,9,10]

Thread.new{
  words.each{ |word| puts( word ) }
}.join

Thread.new{
  numbers.each{ |number| puts( number ) }
}.join
```

At first sight this looks like progress since both threads get the time they need to execute and we haven't had to introduce any unnecessary delays. However, when you take a look at the output you will see that the threads run in sequence – the second thread starts to run after the first thread has finished. But what we really want to do is to get the two threads to run simultaneously, with Ruby switching from one to the next to gives each thread a slice of the available processing time.

The next program, **threads3.rb**, shows one way of achieving this. It creates two threads, as before; however, this time it assigns each thread to a variable: wordsThread and numbersThread:

threads3.rb

```
wordsThread = Thread.new{
  words.each{ |word| puts( word ) }
}
numbersThread = Thread.new{
  numbers.each{ |number| puts( number ) }
}
```

Now it puts these threads into an array and calls the `each` method to pass them into a block where they are received by the block variable, `t`, which simply calls the `join` method on each thread:

```
[wordsThread, numbersThread].each{ |t| t.join }
```

As you will see from the output, the two threads now run 'in parallel' so their output is jumbled up, but there is no artificial delay and the total execution time is negligible.

THREAD PRIORITIES

So far we've given Ruby total freedom in slicing up the time between threads in any way it wishes. But there may be occasions when one thread is more important than the others. For example, if you are writing a file copying program with one thread to do the actual copying and another thread to display the progress bar, it would make sense to give the file copying thread most of the time.

> There may be times when the currently executing thread specifically wants to give execution time to other threads. In principle this is done by calling the `Thread.pass` method. In practice, however, this may not produce quite the results you expect. The `pass` method is discussed in more detail in the **Digging Deeper** section at the end of this chapter.

Ruby lets you assign integer values to indicate the priority of each thread. In theory, threads with higher priorities are allocated more execution time than threads with lower priorities. In practice, things aren't that simple since other factors (such as the order in which threads are run) may affect the amount of time given to each thread. Moreover, in very short programs, the effects of varying the priorities may be impossible to determine. The little words-and-numbers thread example we've used up to now is far too short to show any differences. So let's take a look at a slightly more labour intensive program – one that runs three threads each of which calls a method fifty times in order to compute the factorial of 50.

threads4.rb

```
def fac(n)
  n == 1 ? 1 : n * fac(n-1)
end

t1 = Thread.new{
  0.upto(50) {fac(50); print( "t1\n" )}
}

t2 = Thread.new{
  0.upto(50) {fac(50); print( "t2\n" )}
}

t3 = Thread.new{
  0.upto(50) {fac(50); print( "t3\n" )}
}
```

We can now set specific priorities for each thread:

```
t1.priority = 0
t2.priority = 0
t3.priority = 0
```

In this case, the priorities are the same for each thread so no thread will be given the biggest slice of the action and the results from all three threads will appear in the usual jumble. Now try changing the priority of `t3`:

```
t3.priority = 1
```

This time when you run the code, `t3` will grab most of the time and execute (mostly) before the other threads. The other threads may get a look in at the outset as they are created with equal priorities and the priority is only changed after they have started running. When `t3` has finished, `t1` and `t2` should share the time more or less equally.

So let's suppose you want `t1` and `t2` to run first, sharing time more or less equally and only run `t3` after those two threads have finished. Here's my first attempt; you may want to try it out yourself:

```
t1.priority = 2
t2.priority = 2
t3.priority = 1
```

Hmm, the end result is not what I wanted! It seems that the threads are run in sequence with no time-slicing at all! OK, just for the heck of it, let's try some negative numbers:

```
t1.priority = -1
t2.priority = -1
t3.priority = -2
```

Hurrah! That's more like it. This time, t1 and t2 run concurrently (you may also see t3 executing briefly before the thread priorities are set); then t3 runs. So why do negative values work but positive values don't?

There is nothing special about negative values *per se*. However, you need to bear in mind that every process has at least one thread running – the main thread – and this too has a priority. Its priority happens to be 0.

THE MAIN THREAD PRIORITY

You can easily verify the priority of the main thread:

main_thread.rb

```
puts( Thread.main.priority )                #=> 0
```

So, in the last program (**threads4.rb**), if you set the priority of t1 to 2, it will 'outrank' the main thread itself and will then be given all the execution time it needs until the next thread, t2, comes along and so on. By setting the priorities lower than that of the main thread, you can force the three threads to compete only with themselves since the main thread will always outrank them. If you prefer working with positive numbers, you can specifically set the priority of the main thread to a higher value than all other threads:

```
Thread.main.priority=100
```

threads5.rb

So, now, if we want t2 and t3 to have the same priority, and t1 to have a lower one, we need to set the priorities for those three threads plus the main thread:

```
Thread.main.priority = 200
t1.priority = 0
t2.priority = 1
t3.priority = 1
```

If you look closely at the output it is possible that you may spot one tiny but undesirable side-effect. It is possible (not *certain*, but *possible*) that you will spot some output from the t1 thread right at the outset, just before t2 and t3 kick in and assert their priorities. This is the same problem we noted earlier: each of the threads tries to start running as soon as it is created and t1 may get its own slice of the action before the priorities of the other threads are 'upgraded'. In order to prevent this, we can specifically suspend the thread at the time of creation using Thread.stop like this:

stop_run.rb

```
t1 = Thread.new{
  Thread.stop
  0.upto(50){print( "t1\n" )}
}
```

Now, when we want to start the thread running (in this case, after setting the thread priorities), we call its run method:

```
t1.run
```

MUTEXES

Sometimes two or more threads may each need to access some kind of global resource. This has the potential of producing erroneous results due to the fact that the current state of the global resource may be modified by one thread and this modified value may be unpredictable when used by some other thread. For a simple example, look at this code:

no_mutex.rb

```
$i = 0

a = Thread.new {
  1000000.times{ $i += 1 }
}

b = Thread.new {
  1000000.times{ $i += 1 }
}

a.join
b.join
puts( $i )
```

My intention here is to run two threads, each of which increments the global variable, `$i`, one million times. The expected result of `$i` at the end of this would (naturally) be two million. But, in fact, when I run this, the end value of `$i` is 1088237 (you may see a different result).

The explanation of this is that the two threads are, in effect, competing for access to the global variable, `$i`. This means that, at certain times, thread `a` may get the current value of `$i` (let's suppose it happens to be 100) and simultaneously thread `b` gets the current value of `$i` (still 100). Now, `a` increments the value it just got (`$i` becomes 101) and `b` increments the value *it* just got (so `$i` becomes 101 once again!). In other words, when multiple threads simultaneously access a shared resource, some of them may be working with out-of-date values – that is, values which do not take into account any modifications which have been made by other threads. Over time, errors resulting from these operations accumulate until we end up with results which differ substantially from those we might have anticipated.

To deal with this problem, we need to ensure that, when one thread has access to a global resource, it blocks the access of other threads. This is another way of saying that the access to global resources granted to multiple threads should be 'mutually exclusive'. You can implement this using Ruby's Mutex class, which uses a semaphore to indicate whether or not a resource is currently being accessed, and provides the `synchronize` method to prevent access to resources inside a block. Note that you must require *'thread'* to use the Mutex class. Here is my rewritten code:

mutex.rb

```
require 'thread'
$i = 0

semaphore = Mutex.new

a = Thread.new {
  semaphore.synchronize{
    1000000.times{ $i += 1 }
  }
}

b = Thread.new {
  semaphore.synchronize{
    1000000.times{ $i += 1 }
  }
}

a.join
b.join
puts( $i )
```

This time, the end result of `$i` is 2000000.

Finally, for a slightly more useful example of using Threads take a look at **file_find2.rb**. This sample program uses Ruby's Find class to traverse directories on disk. For a non-threaded example, see **file_find.rb**. Compare this with the **file_info3.rb** program in Chapter 13, which uses the Dir class.

This sets two threads running. The first, `t1`, calls the `processFiles` method to find and display file information (you will need to edit the call to `processFiles` to pass to it a directory name on your system). The second thread, `t2`, simply prints out a message, and this thread runs while `t1` is 'alive' (that is, running or sleeping):

file_find2.rb

```
t1 = Thread.new{
  Thread.stop
  processFiles( '..' ) # edit this directory name
}

t2 = Thread.new{
  Thread.stop
  while t1.alive? do
    print( "\n\t\tProcessing..." )
    Thread.pass
  end
}
```

Each thread yields control using `Thread.pass` (the `t1` thread yields control inside the `processFiles` method). In a real application you could adapt this technique to provide user feedback of some kind while some intensive process (such as directory walking) is taking place.

Digging Deeper

PASSING EXECUTION TO OTHER THREADS

There may be some times at which you specifically want a certain thread to yield execution to any other threads that happen to be running. For example, if you have multiple threads doing steadily updated graphics operations or displaying various bits of 'as it happens' statistical information, you may want to ensure that once one thread has drawn X number of pixels or displayed Y number of statistics, the other threads are guaranteed to get their chances to do something.

In theory, the `Thread.pass` method takes care of this. According to Ruby's source code documentation, `Thread.pass` "Invokes the thread scheduler to pass execution to another thread". This is the example provided by the Ruby documentation:

pass0.rb

```
a = Thread.new {  print "a"; Thread.pass;
                  print "b"; Thread.pass;
                  print "c" }
b = Thread.new {  print "x"; Thread.pass;
                  print "y"; Thread.pass;
                  print "z" }
a.join
b.join
```

According to the documentation, this code, when run, produces this output:

```
axbycz
```

Yes, sure enough, it does. In theory, then, this seems to show that, by calling `Thread.pass` after each call to `print`, these threads pass execution to another thread which is why the output from the two threads alternates.

Being of a suspicious turn of mind, I wondered what the effect would be with the calls to `Thread.pass` removed? Would the first thread hog all the time, only yielding to the second thread when it has finished? The best way to find out is to try it:

pass1.rb

```
a = Thread.new {  print "a";
                  print "b";
                  print "c" }
b = Thread.new {  print "x";
                  print "y";
                  print "z" }
a.join
b.join
```

If my theory is correct (that thread a will hog all the time until it's finished), this would be the expected output:

```
abcdef
```

In fact, (to my surprise!), the output which was actually produced was:

```
axbycz
```

In other words, the result was the same with or without all those calls to Thread.pass. So what, if anything, is Thread.pass doing? And is the documentation wrong when it claims that the pass method "Invokes the thread scheduler to pass execution to another thread"?

For a brief and cynical moment I confess that I toyed with the possibility that the documentation was simply incorrect and that Thread.pass didn't do anything at all. A look into Ruby's C-language source code soon dispelled my doubts; Thread.pass certainly does something but its behaviour is not quite as predictable as the Ruby documentation seems to suggest. Before explaining why this is, let's try out an example of my own:

pass2.rb

```
s = 'start '
a = Thread.new { (1..10).each{
  s << 'a'
  Thread.pass
  }
}
```

```
b = Thread.new { (1..10).each{
  s << 'b'
  Thread.pass
  }
}

a.join
b.join
puts( "#{s} end" )
```

At first sight, this may look very similar to the previous example. It sets two threads running, but instead of printing something out repeatedly these threads repeatedly add a character to a string – 'a' being added by the a thread, 'b' by the b thread. After each operation, Thread.pass passes execution to the other thread. At the end the entire string is displayed. It should come as no surprise that the string contains an alternating sequence of 'a' and 'b':

```
abababababababababab
```

Now, remember that in the previous program, I obtained exactly the same alternating output even when I removed the calls to Thread.pass. Based on that experience, I guess I should expect similar results if I delete Thread.pass in this program. Let's try it:

pass3.rb

```
s = 'start '
a = Thread.new { (1..10).each{
  s << 'a'
  }
}
b = Thread.new { (1..10).each{
  s << 'b'
  }
}

a.join
b.join
puts( "#{s} end" )
```

This time, this is the output:

```
aaaaaaaaaabbbbbbbbbb
```

In other words, this program shows the kind of differing behaviour which I had originally anticipated in the first program (the one I copied out of Ruby's embedded documentation) – which is to say that when the two Threads are left to run under their own steam, the first thread, a, grabs all the time for itself and only when it's finished does the second thread b, get a look in. But by explicitly adding calls to Thread.pass we can force each thread to pass execution to any other threads.

So how can we explain this difference in behaviour? In essence, **pass0.rb** and **pass3.rb** are doing the same things – running two thread and displaying strings from each. The only real difference is that, in **pass3.rb**, the strings are concatenated inside the threads rather than printed. This might not seem like a big deal but it turns out that printing a string takes a bit more time than concatenating one. In effect, then, a call to print introduces a time delay. And as we found out earlier (when we deliberately introduced a delay using sleep), time delays have profound effects on threads.

If you still aren't convinced, try out my rewritten version of **pass0.rb**, which I have creatively named **pass0_new.rb**. This simply replaces the prints with concatenations. Now if you comment and uncomment the calls to **Thread.pass**, you will indeed see differing results:

pass0_new.rb

```
s = ""
a = Thread.new { s << "a"; Thread.pass;
  s << "b"; Thread.pass;
  s << "c" }

b = Thread.new { s << "x"; Thread.pass;
  s << "y"; Thread.pass;
   s << "z" }

a.join
b.join
puts( s )
```

Incidentally, my tests were conducted on a PC running Windows. It is quite possible that different results will be seen on other operating systems. This is because the implementation of the Ruby scheduler – which controls the amount of time allocated to threads – is different on Windows and other operating systems. On Unix the scheduler runs once every 10 milliseconds but on Windows the time sharing is controlled by decrementing a counter when certain operations occur, so that the precise interval is indeterminate.

As a final example, you may want to take a look at the **pass4.rb** program. This creates two threads and immediately suspends them (`Thread.stop`). In the body of each thread the thread's information, including its `object_id` is appended to an array, `arr`, and then `Thread.pass` is called. Finally, the two threads are run and joined and the array, `arr`, is displayed. Try experimenting by uncommenting `Thread.pass` to verify its effect (pay close attention to the execution order of the threads as indicated by their `object_id` identifiers):

pass4.rb

```
arr = []
t1 = Thread.new{
  Thread.stop
  (1..10).each{
    arr << Thread.current.to_s
    Thread.pass
    }
}
t2 = Thread.new{
  Thread.stop
  (1..10).each{ |i|
    arr << Thread.current.to_s
    Thread.pass
    }
}
puts( "Starting threads..." )
t1.run
t2.run
t1.join
t2.join
puts( arr )
```

CHAPTER EIGHTEEN

Debugging and Testing

The development of any real-world application progresses in steps. Most of us would prefer to take more steps forward than backward. In order to minimize the backward steps - caused by coding errors or unforeseen side-effects - we can take advantage of testing and debugging techniques. This chapter aims to provide a brief overview of some of the most useful debugging tools available to Ruby programmers. Bear in mind, however, that if you are using a dedicated Ruby IDE such as Ruby in Steel, you will have more powerful visual debugging tools at your disposal. I will only be discussing the 'standard' tools available to Ruby in this chapter. I won't be discussing tools provided with IDEs.

IRB - INTERACTIVE RUBY

There my be times when you just want to 'try something out' with Ruby. The standard Ruby interpreter, **Ruby.exe**, is far from ideal for this purpose. While it is possible to run **Ruby** from the command prompt and enter bits of code one line at a time, the code will only be executed when you enter an end of file character (CTRL+Z on Windows, CTRL+D on some other operating systems).

For a better way to interact with Ruby, use the Interactive Ruby shell, **IRB.exe**. To start this, go to a command prompt and enter:

```
irb
```

You should now see a prompt similar to the following:

```
irb(main):001:0>
```

Now just start entering some Ruby code. You can enter an expression over more than one line; as soon as the expression is complete, **irb** will evaluate it and display the result. Try out the following (pressing Enter after the **+**):

```
x = ( 10 +
( 2 * 4 ) )
```

When you press Enter after the closing bracket, **irb** will evaluate the expression and show the result:

```
=> 18
```

You can now evaluate x. Enter:

```
x
```

And **irb** shows:

```
=> 18
```

Be careful, though. Try entering this:

```
x = (10
+ (2*4))
```

This time the result is:

```
=> 8
```

This is, in fact, normal Ruby behaviour. It is explained by the fact that a line break acts as a terminator while the + operator, when it begins the new line, acts as a unary operator (it merely asserts that the expression following is positive). You will find a fuller explanation of this in *Digging Deeper* at the end of this chapter. For now, just be aware that when entering expressions a line at a time, the position of the line break is important! When using irb you can tell whether or not the interpreter considers that you have ended a statement. If you have done so, a plain prompt is displayed ending with '>':

```
irb(main):013:1>
```

If the statement is incomplete, the prompt ends with an asterisk:

irb(main):013:1*

In the two examples above, the **>** prompt is shown when you enter the first line without a plus:

```
x = ( 10
```

But the ***** prompt is shown when you enter it with the plus:

```
x = ( 10 +
```

The former case shows that **irb** considers the statement to be complete; the latter case shows that it is waiting for the statement to be completed.

You can, if you wish, load a Ruby program into irb by passing to it the program name, like this:

irb myprogram.rb

You may also invoke it with a variety of options, shown on the next page:

Usage: irb.rb [options] [programfile] [arguments]

-f	Suppress read of ~/.irbrc
-m	Bc mode (load mathn, fraction or matrix are available)
-d	Set $DEBUG to true (same as `ruby -d')
-r load-module	Same as `ruby -r'
-I path	Specify $LOAD_PATH directory
--inspect	Use `inspect' for output (default except for bc mode)
--noinspect	Don't use inspect for output
--readline	Use Readline extension module
--noreadline	Don't use Readline extension module
--prompt	prompt-mode
--prompt-mode	prompt-mode Switch prompt mode. Pre-defined prompt modes are `default', `simple', `xmp' and `inf-ruby'
--inf-ruby-mode	Use prompt appropriate for inf-ruby-mode on emacs. Suppresses --readline.
--simple-prompt	Simple prompt mode
--noprompt	No prompt mode
--tracer	Display trace for each execution of commands.
--back-trace-limit	n Display backtrace top n and tail n. The default value is 16.
--irb_debug n	Set internal debug level to n (not for popular use)
-v, --version	Print the version of irb

You can see a list of these options by entering thus at the command line:

irb --help

You can end an **irb** session by entering the word **quit** at the prompt or by pressing **CTRL+BREAK**.

While **irb** may be useful for trying out some code, it does not provide all the features you need for debugging programs. Ruby does, however, provide a command line debugger.

DEBUGGING

The default Ruby debugger allows you to set breakpoints and watchpoints and evaluate variables while your programs execute. To run a program in the debugger use the **–r debug** option (where `-r` means 'require' and `debug` is the name of the debugging library) when you start the Ruby interpreter. For example, this is how you would debug a program called **debug_test.rb**:

ruby –r debug debug_test.rb

Ubygems? What's Ubygems…?

In some cases if you run the above command, you may see an inscrutable message similar to the following:

c:/ruby/lib/ruby/site_ruby/1.8/ubygems.rb:4:require 'rubygems'

When you then start debugging you will find yourself trying to debug the file '**ubygems.rb**' rather than your program! This seems to be a problem that mainly afflicts Windows users who have installed Ruby using the One-Click Installer:

(*http://rubyforge.org/projects/rubyinstaller/*).

This installer sets up an environment variable, **RUBYOPT=-rubygems**. In most cases this has the desirable effect of allowing your Ruby programs to use the ruby gems 'packaging system' to install Ruby libraries. When you try to use the **–r** option, however, this is interpreted as **–r ubygems** which is why the file **ubygems.rb** is loaded. Ruby conveniently (or possibly confusingly?) provides a file named **ubygems.rb** which does nothing apart from requiring **rubygems.rb**! There are two ways of dealing with this. You can either remove **RUBYOPT** permanently or you can disable it temporarily. If you choose to remove it permanently, however, you may encounter side-effects when using ruby gems later on. To remove it permanently, load *Start Menu, (*then *Settings* if using XP*), Control Panel*; (then *System and Maintenance* if using Vista); click *System* (on Vista, you should now click *Advanced System Settings*); in the System Properties dialog, select the *Advanced* tab; click *Environment Variables*; in the System Variables panel, find **RUBYOPT** and delete it. A safer alternative is to disable the variable at the command prompt prior to loading the debugger. To do this, enter:

```
set RUBYOPT=
```

This will disable the **RUBYOPT** environment variable for this command session only. You can verify this by entering:

```
set RUBYOPT
```

You should see the message:

```
Environment variable RUBYOPT not defined
```

However, open another command window and enter set **RUBYOPT** and you will see that the environment variable here retains its default value.

Once the debugger has started, you can enter various commands to step through your code, set breakpoints to cause the execution to pause as specific lines, set watches to monitor the values of variables and so on. On the next page is a list of the debugging commands available:

b[reak] [file|class:]<line|method> b[reak] [class.]<line|method>	set breakpoint to some position
wat[ch] <expression>	set watchpoint to some expression
cat[ch] <an Exception>	set catchpoint to an exception
b[reak]	list breakpoints
cat[ch]	show catchpoint
del[ete][nnn]	delete some or all breakpoints
disp[lay] <expression>	add expression into display expression list
undisp[lay][nnn]	delete one particular or all display expressions
c[ont]	run until end or breakpoint
s[tep][nnn]	step (into code) 1 line or to line:nnn
n[ext][nnn]	go over one line or till line nnn
w[here]	display frames
f[rame]	alias for where
l[ist][(-|nn-mm)]	list program, - lists backwards nn-mm lists given lines
up[nn]	move to higher frame
down[nn]	move to lower frame
fin[ish]	return to outer frame
tr[ace] (on|off)	Set trace mode of current thread
tr[ace] (on|off) all	Set trace mode of all threads
q[uit]	exit from debugger
v[ar] g[lobal]	show global variables
v[ar] l[ocal]	show local variables
v[ar] i[nstance] <object>	show instance variables of object
v[ar] c[onst] <object>	show constants of object
m[ethod] i[nstance] <obj>	show methods of object
m[ethod] <class|module>	show instance methods of class or module
th[read] l[ist]	list all threads
th[read] c[ur[rent]]	show current thread
th[read] [sw[itch]] <nnn>	switch thread context to nnn
th[read] stop <nnn>	stop thread nnn
th[read] resume <nnn>	resume thread nnn
p expression	evaluate expression and print its value
h[elp]	print this help
<everything else>	Evaluate

Let's see how a few of these commands might be used in a real debugging session. Open a system prompt and navigate to the directory containing the file, **debug_test.rb**. Start the debugger by entering:

debug_test.rb

ruby –r debug debug_test.rb

Now, let's try out a few commands. In these example, I've written *[Enter]* to show that you should press the Enter key after each command. First let's see a code listing:

l [Enter]

This shows the first few lines of the program. The **l** (lowercase 'L') or **list** command lists the code in bite-sized chunks. The actual number of lines will vary with the code being debugged. List some more:

l [Enter]
l [Enter]

Or list a specific number of lines(here the letter 'l' followed by the digit 1, a hyphen and 100):

l 1-100 [Enter]

Let's put a breakpoint on line 78:

b 78 [Enter]

The Ruby debugger should reply:

Set breakpoint 1 at debug_test.rb:78

We might also set one or more watchpoints. A watchpoint can be used to trigger a break on a simple variable (e.g. entering **wat @t2** would break when the `@t2` object is created); or it may be set to match a specific value (e.g. **i == 10**). Here I want to set a watchpoint that breaks when the `name` attribute of `@t4` is "wombat"):

```
wat @t4.name == “wombat” [Enter]
```

The debugger should confirm this:

```
Set watchpoint 2:@t4.name == “wombat”
```

Notice the watchpoint number is 2. You’ll need that number if you subsequently decide to delete the watchpoint. OK, so now let’s continue execution:

```
c [Enter]
```

The program will run until it hits the breakpoint. You will see a message similar to the following:

```
Breakpoint 1, toplevel at debug_test.rb:78
debug_test.rb:78:        puts( “Game start” )
```

Here it shows the line number it’s stopped on and the code on that line. Let’s continue:

```
c [Enter]
```

This time it breaks here:

```
Watchpoint 2, toplevel at debug_test.rb:85
debug_test.rb:85:        @t5 = Treasure.new("ant", 2)
```

This is the line immediately following the successful evaluation of the watch-point condition. Check that by listing the line number indicated:

```
l 85
```

The debugger shows a set of lines with the current line of execution (86) high-lighted:

```
[80, 89] in debug_test.rb
  80    #   i) Treasures
  81    @t1 = Treasure.new("A sword", 800)
  82    @t4 = Treasure.new( "potto", 500 )
  83    @t2 = Treasure.new("A dragon Horde", 550)
  84    @t3 = Treasure.new("An Elvish Ring", 3000)
  85    @t4 = Treasure.new("wombat", 10000)
=> 86    @t5 = Treasure.new("ant", 2)
  87    @t6 = Treasure.new("sproggit", 400)
  88
  89    #   ii) Rooms
```

As you can see, line 85 contains the code that matches the watchpoint condition. Notice that execution did not stop after line 82, where `@t4` was originally created, as the watchpoint condition was not met there (its `name` attribute was "potto", not "wombat"). If you want to inspect the value of a variable when paused at a breakpoint or watchpint, just enter its name. Try this:

@t4 [Enter]

The debugger will display:

```
#<Treasure:0x315617c @value=10000, @name="wombat">
```

You can equally enter other expressions to be evaluated:

@t1.value [Enter]
10+4/2 [Enter]

Now delete the watchpoint (recall that its number is 2):

del 2 [Enter]

And continue until the program exists:

c [Enter]

There are many more commands which can be used to debug a program in this way and you may want to experiment with those shown in the table above. You can also view a list of commands during a debugging session by entering **help** or just **h**:

h [Enter]

To quit a debugging session, enter **quit** or **q**:

q [Enter]

While the standard Ruby debugger has its uses, it is far from as simple or convenient to use as one of the graphical debuggers provided by integrated development environments. Moreover, it is quit slow. In my view, it is fine for debugging simple scripts, but cannot be recommended for debugging large and complex programs.

UNIT TESTING

Unit Testing is a sort of post-debugging testing technique which lets you try out bits of your program in order to verify that they work as expected. The basic idea is that you can write a number of 'assertions' stating that certain results should be obtained as the consequence of certain actions. For example, you might assert that the return value of a specific method should be 100 or that it should be a Boolean or that it should be an instance of a specific class. If, when the test is run, the assertion proves to be correct, it passes the test; if it is incorrect, the test fails.

Here's an example, that will fail if the `getVal` method of the object, `t`, returns any value other than 100:

```
assert_equal(100, t.getVal)
```

But you can't just pepper your code with assertions of this sort. There are precise rules to the game. First you have to require the **test/unit** file. Then you need to derive a test class from the TestCase class which is found in the Unit module which is itself in the Test module:

```
class MyTest < Test::Unit::TestCase
```

Inside this class you can write one or more methods, each of which constitutes a test containing one or more assertions. The method names must begin with `test` (so methods called `test1` or `testMyProgram` are ok, but a method called `myTestMethod` isn't). This is a test containing the single assertion that the return value of `TestClass.new(100).getVal` is 1000:

```
def test2
  assert_equal(1000,TestClass.new(100).getVal)
end
```

And here is a complete (albeit simple) test suite in which I have defined a Test-Case class called MyTest which tests the class, TestClass. Here (with a little imagination!), TestClass may be taken to represent a whole program that I want to test:

test1.rb

```
require 'test/unit'

class TestClass
  def initialize( aVal )
    @val = aVal * 10
  end

  def getVal
    return @val
  end
end

class MyTest < Test::Unit::TestCase
  def test1
    t = TestClass.new(10)
    assert_equal(100, t.getVal)
    assert_equal(101, t.getVal)
    assert(100 != t.getVal)
  end

  def test2
    assert_equal(1000,TestClass.new(100).getVal)
  end
end
```

This test suite contains two tests: `test1` (which contains three assertions) and `test2` (which contains one). In order to run the tests, you just need to run the program; you don't have to create an instance of MyClass.

You will see a report of the results which states that there were two tests, three assertions and one failure. In fact, I made four assertions. However, assertions following a failure are not evaluated in a given test. In test1, this assertion fails:

```
assert_equal(101, t.getVal)
```

Having failed, the next assertion is skipped. If I now correct this (asserting 100 instead of 101, the next assertion will also be tested:

```
assert(100 != t.getVal)
```

This too fails. This time the report states that four assertions have been evaluated with one failure. Of course, in a real-life situation, you should aim to write correct assertions and when any failures are reported, it should be the failing code that is rewritten – not the assertion!

For a slightly more complex example of testing, see the **test2.rb** program (which requires a file called buggy.rb). This is a small adventure game which includes the following test methods:

test2.rb

```
def test1
    @game.treasures.each{ |t|
      assert(t.value < 2000, "FAIL: #{t} t.value = #{t.value}" )
    }
  end

  def test2
    assert_kind_of( TestMod::Adventure::Map, @game.map)
    assert_kind_of( Array, @game.map)
   end
```

Here the first method performs an assert test on an array of objects passed into a block and it fails when a value attribute is not less than 2000. The second method tests the class types of two objects using the assert_kind_of method. The second test in this method fails when @game.map is found to be of the type, Test-Mod::Adventure::Map rather than Array as is asserted.

The code also contains two more methods named `setup` and `teardown`. When defined, methods with these names will be run before and after each test method. In other words, in **test2.rb**, the following methods will run in this order: `setup`, `test1`, `teardown`, `setup`, `test2`, `teardown`. This gives you the opportunity of reinitializing any variables to specific values prior to running each test or, as in this case, recreating objects to ensure that they are in a known state:

```
def setup
  @game = TestMod::Adventure.new
end

def teardown
  @game.endgame
end
```

Digging Deeper

ASSERTIONS AVAILABLE WHEN UNIT TESTING

assert(boolean, message=nil)
  Asserts that boolean is not false or nil.

assert_block(message="assert_block failed.") {|| ...}
  The assertion upon which all other assertions are based. Passes if the block yields true.

assert_equal(expected, actual, message=nil)
  Passes if expected == +actual.

assert_in_delta(expected_float, actual_float, delta, message="")
  Passes if expected_float and actual_float are equal within delta tolerance.

assert_instance_of(klass, object, message="")
  Passes if object .instance_of? klass

assert_kind_of(klass, object, message="")
  Passes if object .kind_of? klass

assert_match(pattern, string, message="")
  Passes if string =~ pattern.

assert_nil(object, message="")
  Passes if object is nil.

assert_no_match(regexp, string, message="")
  Passes if regexp !~ string

assert_not_equal(expected, actual, message="")
  Passes if expected != actual

assert_not_nil(object, message="")
  Passes if ! object .nil?

assert_not_same(expected, actual, message="")
  Passes if ! actual .equal? expected

assert_nothing_raised(*args) {|| ...}
  Passes if block does not raise an exception.

assert_nothing_thrown(message="", &proc)
  Passes if block does not throw anything.

assert_operator(object1, operator, object2, message="")
  Compares the +object1+ with +object2+ using operator.
  Passes if object1.send(operator, object2) is true.

assert_raise(*args) {|| ...}

Passes if the block raises one of the given exceptions.

`assert_raises(*args, &block)`

Alias of `assert_raise`.
(Deprecated in Ruby 1.9, and to be removed in 2.0).

`assert_respond_to(object, method, message="")`

Passes if object `.respond_to?` method

`assert_same(expected, actual, message="")`

Passes if actual `.equal?` expected (i.e. they are the same instance).

`assert_send(send_array, message="")`

Passes if the method send returns a true value.

`assert_throws(expected_symbol, message="", &proc)`

Passes if the block throws expected_symbol

`build_message(head, template=nil, *arguments)`

Builds a failure message. head is added before the template and arguments replaces the '?'s positionally in the template.

`flunk(message="Flunked")`

`flunk` always fails.

LINE BREAKS ARE SIGNIFICANT

I said earlier that you need to take care when entering line breaks into the interactive Ruby console (IRB) since the position of line breaks may alter the meaning of your Ruby code. So for example, this:

linebreaks.rb

```
x = ( 10 +
( 2 * 4 ) )
```

... assigns `18` to `x`, but this:

```
x = (10
+ (2*4))
```

...assigns `8` to `x`.

This is not merely a quirk of IRB. This is normal behaviour of Ruby code even when entered into a text editor and executed by the Ruby interpreter. The second example shown above evaluates 10, finds it to be a perfectly acceptable value and promptly forgets about it; then it evaluates + (2*4) which it also finds to be an acceptable value (8) but which has no connection with the previous value (10), and so 8 is returned and assigned to x.

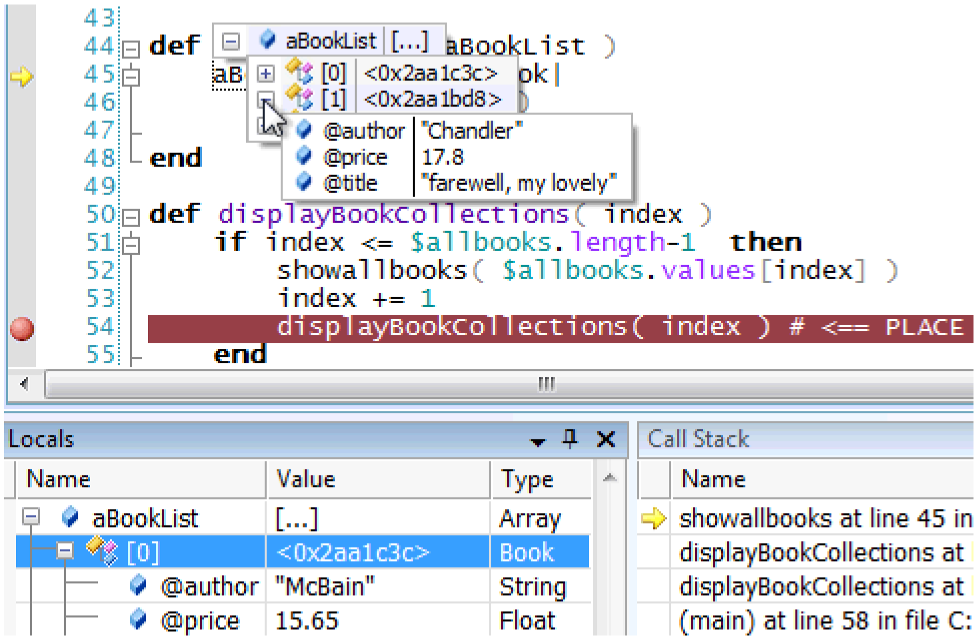
If you want to 'tie lines together' by telling Ruby to evaluate expressions split over multiple lines, ignoring the line breaks, you can use the line continuation character \. This is what I've done here:

```
x = (10 \
+ (2*4) )
```

This time, x is assigned the value 18.

GRAPHICAL DEBUGGERS

For serious debugging, I would strongly recommend a graphical debugger. For example the debugger in the Ruby In Steel IDE allows you to set breakpoints and watchpoints by clicking the margin of the editor. It lets you monitor the values of selected 'watch variables' or all local variables in separate docked windows. It maintains a 'callstack' of all method calls leading to the current point of execution and allows you to navigate 'backwards' through the callstack to view the changing values of variables. It also has full 'drill-down' expansion of variables to allow you to expand Arrays and hashes and look 'inside' complex objects. These capabilities go well beyond the features of the standard Ruby debugger.



```
def            aBookList )
    aB           ok|
              )

end

def displayBookCollections( index )
    if index <= $allbooks.length-1  then
        showallbooks( $allbooks.values[index] )
        index += 1
        displayBookCollections( index ) # <== PLACE
    end
```

aBookList	[...]	
[0]	<0x2aa1c3c>	
[1]	<0x2aa1bd8>	
@author	"Chandler"	
@price	17.8	
@title	"farewell, my lovely"	

Locals

Name	Value	Type
aBookList	[...]	Array
[0]	<0x2aa1c3c>	Book
@author	"McBain"	String
@price	15.65	Float

Call Stack

Name
showallbooks at line 45 in
displayBookCollections at
displayBookCollections at
(main) at line 58 in file C:

The Ruby In Steel debugger

CHAPTER NINETEEN

Ruby On Rails

Rails has become so closely connected with Ruby that it is now quite commonplace for people to talk about programming "in" Ruby On Rails as though "Ruby On Rails" were the name of the programming language.

In fact, Rails is a framework – a set of tools and code libraries – which can be used in cahoots with Ruby. It helps in the creation of database-driven applications with a web-based front end. Rails gives you the ability to develop web sites which respond to user interaction: for example, users might be able to enter and save data on one page and search for and display existing data on other pages. This makes Rails suitable for creating dynamic web sites which generate web pages 'on the fly' rather than loading up static, pre-designed pages. Typical applications include: collaborative sites such as online communities, multi-author books or Wikis, shopping sites, discussion forums and blogs.

We'll go through a hands-on guide to creating a blog shortly. First though, let's take a closer look at the nitty-gritty details of the Rails framework.

This chapter aims to give you a 'taste' of developing in Ruby On Rails. Bear in mind, however, that Rails is a big and complex framework. We shall be covering only the bare essentials! We are using version 2 of Rails in this chapter. The examples will not work with Rails 1 and are not guaranteed to work with any future Rails versions.

FIRST INSTALL RAILS

A) 'DO IT YOURSELF'...

Rails is not a standard part of Ruby so you will have to install it as a separate operation. There are various ways in which you can do this. The easiest way is to use an all-one installer (some alternatives are described below). However, you may also install Rails and the tools it requires one at a time. Rails may be installed using the Ruby Gem 'package manager'. As long as you are connected to the Internet, this will go online to find and install the latest version of Rails. At the command prompt, enter:

```
gem install rails
```

Alternatively you may download and install Rails from the Ruby On Rails web site, **http://www.rubyonrails.org**. Most Rails applications require a database too. You will need to install a database as a separate operation. The free MySQL database server is widely used for this purpose. You can find basic installation help on MySQL in the Appendix.

B) OR USE AN 'ALL IN ONE' INSTALLER...

Mac users can user the Locomotive installer to setup a Rails environment with added utilities (**http://sourceforge.net/projects/locomotive**). Windows users can save themselves some effort by using the InstantRails installer or the Ruby In Steel All-in-One installer. InstantRails installs Ruby, Rails, the Apache web server and MySQL. Everything is put into a separate directory structure so should, in principle, be able to cohabit with any other Ruby installations on the same PC without causing any unforeseen side-effects. In fact, I've found that InstantRails can sometimes be problematic if you already have a separate installation of Apache. Otherwise, it provides an easy way to get up and running with Rails.

Download InstantRails from:

http://instantrails.rubyforge.org/wiki/wiki.pl

The Ruby In Steel All-in-One installer installs Ruby, Gems, Rails, WEBrick, MySQL plus (optionally) a free edition of Microsoft Visual Studio. You may also install either a free or commercial (or trial) edition of the Ruby In Steel IDE. The examples in this chapter have all been tested using the Ruby In Steel 'all in one installer' default configuration.

Download the Ruby In Steel All-in-One installer from:

http://www.sapphiresteel.com/spip?page=download

MVC – MODEL, VIEW, CONTROLLER

A Rails application is divided into three main areas – the Model, the View and the Controller. Put simply, a Model is the data part – the database and any programmatic operations (such as calculations) which are done upon that data; the View is what the end user sees – in Rails terms that means the web pages that appear in the browser; and the Controller is the programming glue that joins the Model to the View.

The Model-View-Controller methodology is used, in various forms by all kinds of programming languages and frameworks. It is more fully described in the *Digging Deeper* section. For the sake of brevity, I shall henceforward call it MVC.

A FIRST RUBY ON RAILS APPLICATION

Without more ado, let's start programming with Rails. I'll assume that you have Rails installed, along with a web server. I happen to be using the WEBrick server but you may use some other server such as LightTPD or Mongrel. More information on web servers can be found in the Appendix.

This chapter assumes that you are using only the 'raw' Rails development tools plus, at the very least, a text editor and a web browser; as a consequence, you will find that you frequently have to start up separate programs and enter commands at the system prompt. An integrated development environment should allow you to accomplish these tasks much more easily.

Note: Unlike the plain Ruby code examples in this book, I have not supplied all the code for the Ruby On Rails applications described in this chapter. There are three reasons for this:

1) Each Rails application comprises a great many files and folders.
2) I would also have to supply data for each database and you would have to import it prior to using it.
3) it is not only simpler to create your own Rails applications yourself but doing so will also help you understand how Rails works. I have, however, supplied some sample files – component parts of a complete application – with which you can compare your own code in case you run into problems.

CREATE A RAILS APPLICATION

For the sake of simplicity, this first application will not use a database at all. This will let us explore the View and the Controller without having to worry about the complexities of the Model.

To begin, open a system prompt (on Windows, select *Start Menu,* and enter **cmd** into the *Run* or *Search* box). Navigate to a directory into which you intend to place your Rails applications. Let's assume this is **C:\railsapps**. Check that Rails is installed and on the system path. Enter:

```
rails
```

All being well, you should now see a screenful of help about using the **rails** command. If not, there is a problem with your Rails installation which you need to fix before continuing.

Assuming Rails is working, you can now create an application. Enter this:

```
rails helloworld
```

After a bit of whirring of your hard disk, you should see a list of the files which Rails has just created:

```
 create  app/controllers
 create  app/helpers
 create  app/models
 create  app/views/layouts
 create  config/environments
…(etcetera)
```

Take a look at these files using your computer's file manager (e.g. the Windows Explorer). Beneath the directory in which you ran the Rails command (**\helloworld**), you will see that several new directories have been created: **\app**, **\config**, **\db** and so on. Some of these have subdirectories. The **\app** directory, for example, contains **\controllers**, **\helpers**, **\models** and **\views**. The **\views** directory itself contains a subdirectory, **\layouts** and so on.

The directory structure in a Rails application is far from random; the directories (or 'folders') and the names of the files they contain define the relationships between the various parts of the application. The idea behind this is that, by adopting a well-defined file-and-folder structure, you can avoid the necessity of writing lots of configuration files to link the various bits of the application together. There is a simplified guide to the default directory structure of rails in *Digging Deeper* at the end of this chapter.

Now, at the system prompt, change directory to the top-level folder (**\helloworld**) of your newly generated Rails application. Assuming you are still in the **C:\railsapps** directory and you named the Rails application **helloworld**, as suggested earlier, you would (on Windows) enter this command to change to that directory:

```
cd helloworld
```

Now run the server. If (like me) you are using WEBrick, you should enter:

```
ruby script/server
```

Note that other servers may be started in different ways and, if the above does not work, you will need to consult the server's documentation. In the above example, **script** is a directory which was created when you ran the **rails** command and **server** is the name of the code file which runs the WEBrick server. You should now see something similar to the following:

```
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2006-11-20 13:46:01] INFO  WEBrick 1.3.1
[2006-11-20 13:46:01] INFO  ruby 1.8.4 (2005-12-24) [i386-mswin32]
[2006-11-20 13:46:01] INFO  WEBrick::HTTPServer#start: pid=4568 port=3000
```

Problems...?

If, instead of the above output, you see error messages, check that you have entered the server command exactly as shown from the appropriate directory (**helloworld**):

```
ruby script/server
```

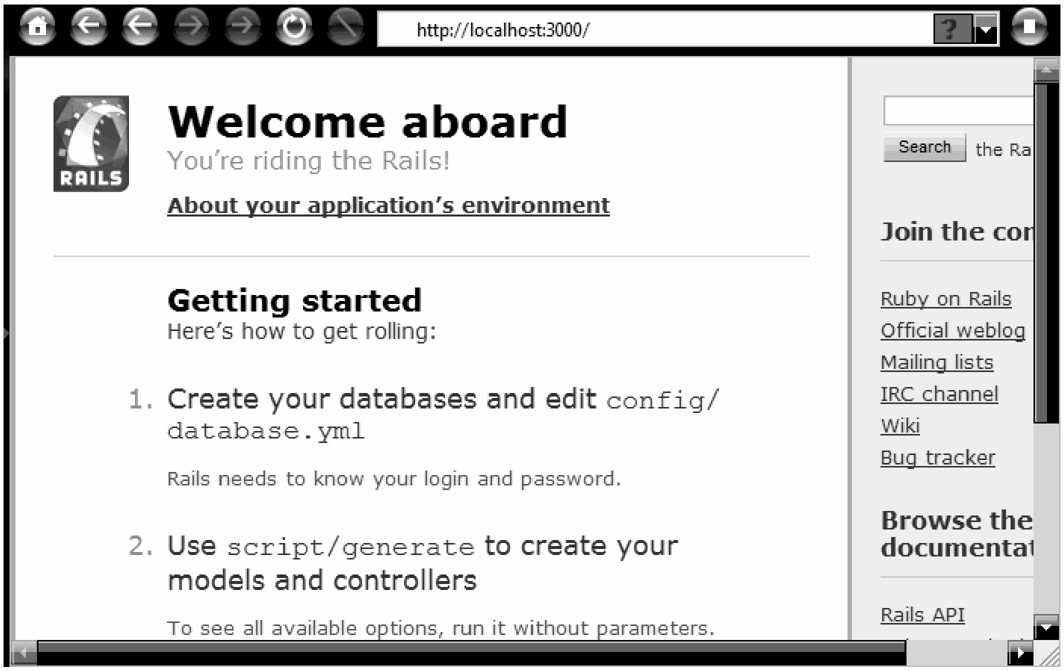
If you still have problems, it is possible that the default port (3000) is already in use – for example, if you already have an Apache server installed on the same PC. In that case, try some other value such as 3003, placing this number after **-p** when you run the script:

```
ruby script/server –p3003
```

Now fire up a web browser. Enter the host name, followed by a colon and the port number into its address bar. The host name should (normally) be *localhost* and the port number should match the one used when starting the server or else it defaults to 3000. Here is an example:

http://localhost:3000/

The browser should now display a page welcoming you aboard Rails. If not, verify that your server is running on the port specified in the URL.


http://localhost:3000/

Welcome aboard

You're riding the Rails!

About your application's environment

Getting started

Here's how to get rolling:

1. Create your databases and edit config/database.yml

Rails needs to know your login and password.

2. Use script/generate to create your models and controllers

To see all available options, run it without parameters.

CREATE A CONTROLLER

As mentioned earlier, a Controller is where much of your Ruby code will live. It is the part of the application that sits between the View (what appears in the browser) and the Model (what happens to the data). As this is a 'Hello world' application, let's create a Controller to say 'hello'. In the spirit of originality, I'll call this the **SayHello** controller. Once again, you can create this by running a script at the system prompt. You will need to open another command window in the directory from which you previously ran the server script (for example, **C:\railsapps\helloworld**). You can't re-use your existing command window as the server is running in that one and you would need to close it down to get back to the prompt – and that would stop our Rails application from working!

At the prompt enter this (use the capitalization of **SayHello** as shown):

```
ruby script/generate controller SayHello
```

After a few moments, you will be informed that the following files and directories have been created (the script will also tell you that some directories already exist and are, therefore, not created):

app/views/say_hello
app/controllers/say_hello_controller.rb
test/functional/say_hello_controller_test.rb
app/helpers/say_hello_helper.rb

Note: The **generate controller** script also creates one other Ruby file, **application.rb**, which is the controller for the entire application, plus a folder, **/views/say_hello**, which we shall make use of shortly.

Notice how Rails has parsed the name SayHello into two lowercase words, *say* and *hello*, separated by an underscore and it has used this name as the first part of three separate Ruby files. This is just one example of the 'configuration by convention' approach which Rails uses.

Here the Ruby files **say_hello_controller_test.rb** and **say_hello_helper.rb** are supplied as repositories (should you so wish) for testing and utility ('helper') code respectively. More important, though, is the controller file itself, **say_hello_controller.rb,** which has been created in **\helloworld\app\controllers**. Open this file in a text editor. This empty method has been auto-generated:

```
class SayHelloController < ApplicationController
end
```

Inside this class we can write some code to be executed when a certain page is displayed. Edit the class definition to match the following:

```
class SayHelloController < ApplicationController
  def index
    render :text => "Hello world"
  end

  def bye
    render :text => "Bye bye"
  end
end
```

This now contains two methods, `index` and `bye`. Each method contains a single line of code. In spite of the fact that I have omitted brackets (a bracket-light style of coding is favoured by many Rails developers), you can probably deduce that `render` is a method which takes a Hash as an argument; the Hash itself containing a key-value pair comprising a symbol and a string. For bracket-lovers, the `index` method can be rewritten like this:

```
def index
  render( { :text => "Hello world" } )
end
```

And there you have your first real Rails application. To try it out you need to go back to the web browser and enter the full 'address' of the two functions you just wrote. But first you may need to restart your server. Just press CTRL+C in the command window where the server is running. When the server exists, restart by entering:

```
ruby script/server
```

Now enter an address to access a controller method. The address takes the form of the host and port (the same as you entered previously – for example, **http://localhost:3000**), plus the name of the controller (**/say_hello**) and finally the name of a specific method (**/index** or **/bye**). Try entering these into your browser's address field, once again ensuring that you use the appropriate port number if it is not 3000:

```
http://localhost:3000/say_hello/index
http://localhost:3000/say_hello/bye
```

Your browser should display "Hello world" and "Bye bye" respectively for each address. If all is working at this point, you can move on to the section headed *'Anatomy Of A Simple Rails Application'*. However, if you are seeing database errors, read the next section, *'Can't Find The Database?'*, first...

CAN'T FIND THE DATABASE?

If you choose to use the SQLite3 database, you should first install it following the instructions here:

http://wiki.rubyonrails.org/rails/pages/HowtoUseSQLite

If (like me) you have decided to use the MySQL database and, assuming that MySQL is correctly installed, (see *Appendix*), it is nevertheless possible that Rails will display an error message similar to the following when you try to run your application:

```
no such file to load -- mysql
```

Some versions of Rails (for example, Rails 2.2) require that the MySQL gem be installed as a separate operation. To do this, at the system prompt, enter:

```
gem install mysql
```

On Windows, when you now run your application it is possible that you will see a different error message similar to this:

```
The specified module could not be found.
c:/ruby/lib/ruby/gems/1.8/gems/mysql-2.7.3-x86-mswin32/ext/mysql.so
```

If you encounter this problem, you should be able to fix it by making a copy of a file called **libmySQL.dll** from your MySQL binary directory (for example, *C:\Program Files\MySQL\MySQL Server 5.0\bin*) into the Ruby binary directory (for example, *C:\ruby\bin*). Restart your application (shut down and restart the server), then try running it again.

Our simple 'hello world' application does not require a database. Verify that the database adapter (*sqlite3* or *mysql*, for example) is correctly specified but that no database name is given in the 'development' section of the database configuration file, *\app\config\database.yml*.

As I am using MySQL, my entry is as follows (where '*root*' is my MySQL user name and '*mypassword*' is my MySQL password:

```
development:
  adapter: mysql
  host: localhost
  username: root
  database:
  password: mypassword
```

ANATOMY OF A SIMPLE RAILS APPLICATION

Rails uses the **index** method as a default so you can omit that part of the URL when entering the address into the browser:

http://localhost:3000/say_hello

Before moving on, let's take a closer look at the class we are using. Rails has named the class by appending Controller to the name which we specified when running the controller generator script (HelloWorld) and it has made it a descendant of the ApplicationController class:

```
class SayHelloController < ApplicationController
```

So what exactly is the ApplicationController class? Now, you may recall that I mentioned that the **generate/controller** script which we ran earlier silently created a file called **application.rb** inside the **/app/controllers** folder. This file is the application controller and, if you open it, you will see that it contains a class called:

```
ApplicationController < ActionController::Base
```

So, our SayHelloController class descends from the ApplicationController class which itself descends from the Base class in the ActionController module. You can prove this by climbing back through the hierarchy and asking each class to display itself.

This, incidentally, also gives us the chance to try doing some real Ruby programming in the SayHelloController class.

Just edit the contents of **say_hello_controller.rb** file to match the following (or copy and paste the code from the **sayhello1.rb** file in the code archive for this chapter):

sayhello1.rb

```
class SayHelloController < ApplicationController
  def showFamily( aClass, msg )
    if (aClass != nil) then
      msg += "<br />#{aClass}"
      showFamily( aClass.superclass, msg )
    else
      render :text => msg
    end
  end

  def index
    showFamily( self.class, "Class Hierarchy of self..." )
  end
end
```

To see the result, enter this address into your browser (once again, change the port number if necessary):

http://localhost:3000/say_hello/

Your web browser should now display...

```
Class Hierarchy of self...
SayHelloController
ApplicationController
ActionController::Base
Object
```

As you can see, a Rails controller file contains Ruby code. You can use all the usual Ruby classes in a controller such as strings and Hashes and you can call methods and pass arguments.

But bear in mind that the end result needs to be displayed in a web page. This has certain consequences. For example, instead of putting linefeeds "\n" into strings, you should use HTML paragraph, <P> or break <br /> tags and it is only permissible to call render once each time a page is displayed, which explains why I've constructed a string in the course of calling the method recursively and then passed this to the render method right at the end:

```
def showFamily( aClass, msg )
    if (aClass != nil) then
      msg += "<br />#{aClass}"
      showFamily( aClass.superclass, msg )
    else
     render :text => msg
    end
end
```

THE GENERATE CONTROLLER SCRIPT SUMMARIZED

Let's briefly go over what's happening when we run the **generate controller** script. Each time a new controller is generated it creates a Ruby code file in the **app/controllers** directory, with a name matching the name you entered but all in lowercase with any non-initial capitals which you specified being preceded by an underscore and **_controller** appended. So, if you entered **SayHello** the controller file will be called **say_hello_controller.rb**). The controller will contain a class definition such as: **SayHelloController**. You may also specify '*views*' such as **index** and **bye** by including those view names when you execute the script…

```
ruby script/generate controller SayHello index bye
```

In this case, the Controller class will automatically be provided with methods with names matching those views (`def index` and `def bye`). At any rate, whether or not you specify views, a folder in the **/views** directory is created with a name matching the controller (**views/say_hello**). In fact, the script also creates a few other files including some more Ruby files in the **/helpers** folder, but we can ignore these for now.

If you specified view names when running the controller script, some files with matching names and the extension **.html.erb** will be added to the appropriate view folder. For instance, if you entered the command…

```
ruby script/generate controller SayHello xxx
```

…the **/views/say_hello** directory should now contain a file called **xxx.html.erb**. If, on the other hand, you entered…

```
ruby script/generate controller Blather xxx bye snibbit
```

…the **views/blather** directory should now contain three files:

xxx.html.erb, bye.html.erb and **snibbit.html.erb**

CREATE A VIEW

While it would be, just about, possible to create an entire application by coding everything inside a Controller, you would end up with some pretty ugly web pages. In order to apply more formatting you need to create a View.

You can think of a View as a HTML page that will be displayed when someone logs onto a specific web address – in which case, the name of the View forms the final part of the address as in the previous examples where the **/index** and **/bye** parts of the URL took us to views which displayed data supplied by the `index` and `bye` methods in the controller.

You may create HTML view 'templates' which match these web addresses and the corresponding method names. Using a HTML (or plain text) editor, create a file named **index.html.erb** in the **\app\views\say_hello** directory.

Remember, you can optionally create one or more view templates when you initially generate the controller. This is done by appending several names to the end of the command used to run the script to generate the controller:

ruby script/generate controller Blather index bye snibbit

This script would create the *Blather* controller and three views: *index*, *bye* and *snibbit*.

Now that we have a view template we can edit it in order to control the way that data is displayed in the web page. That means we won't need display plain, unformatted text using the `render` method in the controller from now on. But, with the view being out of the controller's control (so to speak), how can the controller pass data to the view? In turns out that it can do this by assigning data to an instance variable.

Edit the code in **say_hello_controller.rb** (or delete it and paste in my code from the file, **sayhello2.rb**) so that it matches the following:

sayhello2.rb

```
class SayHelloController < ApplicationController
  def showFamily( aClass, msg )
    if (aClass != nil) then
      msg += "<li>#{aClass}</li>"
      showFamily( aClass.superclass, msg )
    else
      return msg
    end
  end

  def index
    @class_hierarchy = "<ul>#{showFamily( self.class, "" )}</ul>"
  end
end
```

This version calls the showFamily() method in order to build up a string inside two HTML 'unordered list' tags, <ul> and </ul>. Each time a class name is found it is placed between two HTML 'list item' tags <li> and </li>. The complete string forms a valid HTML fragment and the index method simply assigns this string the a variable named @class_hierarchy.

HTML Tags In The Controller...?

Some Ruby On Rails developers object to having *any* HTML tags, no matter how trivial, included in the Controller code. In my opinion, if you intend to display the end results in a web page, it little matters where you put the odd <p>, <ul> or <li> tag. While the MVC paradigm encourages strong separation between the program code of the Controller and the layout definition of the View, you will inevitably have to make some compromises - at the very least by putting some program code into the View. Avoiding the use of HTML tags in the Controller is, largely, an aesthetic rather than a pragmatic objection. I personally have no very strong views on the subject, though (be warned!) other people do...

All we need to do now is to find some way of putting that HTML fragment into a fully formed HTML page. That's where the View comes in. Open up the view file which you just created, **index.html.erb**, in the **app/views/say_hello** folder. According to the Rails naming convention - this is the default view (the 'index' page) which is partnered with the **say_hello_controller.rb** file. Since Rails works out relationships based on file, folder, class and method names, we don't have to load or require any files by name or write any configuration details.

In the **index.html.erb** file add this:

```
<h1>This is the Controller's Class Hierarchy</h1>
<%= @class_hierarchy %>
```

The first line is nothing more than plain HTML formatting which defines the text enclosed by the `<h1></h1>` tags as a heading. The next line is more interesting. It contains the variable, `@class_hierarchy`. Look back at the `index` method in **say_hello_controller.rb** and you'll see that this is the variable to which we assigned a string. Here in the View, `@class_hierarchy` is placed between two odd-looking delimiters `<%=` and `%>`. These are special Rails tags. They are used to embed bits of Ruby which will be executed prior to displaying a web page in the browser. The page that is finally displayed will be a fully-formed HTML page which includes any HTML fragments from the view template file plus the results, after execution, of any embedded Ruby code. Try it out now, by entering the page address into your browser:

http://localhost:3000/say_hello/

This should now display the heading "This is the Controller's Class Hierarchy" in big, bold letters followed by a list of classes each element of which is preceded by a dot:

- SayHelloController
- ApplicationController
- ActionController::Base
- Object

You could, if you wish, remove all the HTML from the view file by creating the heading in the controller and assigning the resulting string to another variable. You can do this by editing the *index* method in **say_hello_controller.rb** to this:

```
def index
  @heading = "<h1>This is the Controller's Class Hierarchy</h1>"
  @class_hierarchy = "<ul>#{showFamily( self.class, "" )}</ul>"
end
```

Then edit the view file (**/app/views/say_hello/index.html.erb**) to this:

say_hello.html.erb

```
<%= @heading %>
<%= @class_hierarchy %>
```

If you do this, the end result, as displayed in the web page, will remain unchanged.

RAILS TAGS

There are two variations on the Rails tags which you can place into Rails HTML 'Embedded Ruby' (*ERb*) template files. The ones we've used so far include an equals sign at the beginning, <%=.

These tags cause Rails not only to evaluate Ruby expressions but also to display the results in a web page. If you omit the equals sign in the opening delimiter <% then the code will be evaluated but the result will not be displayed.

> **ERb** ('Embedded Ruby') files contain a mix of HTML markup and Ruby code between tags such as <%= and %>. Rails processes these files before the final pages are displayed in the web browser, executing the embedded Ruby and constructing a HTML page as a result.

If you wish, you can place quite long pieces of code – your entire Ruby program even! - between <% and %> tags and then use <%= and %> when you want to display something in the web page. In fact, we could rewrite our application by omitting the controller entirely and putting everything into the view. Try it by editing **app/views/say_hello/index.html.erb** to match the following (or copying and pasting the code from the same file, **embed_ruby.html.erb**):

embed_ruby.rhtml

```
<% def showFamily( aClass, msg )
    if (aClass != nil) then
      msg += "<li>#{aClass}</li>"
      showFamily( aClass.superclass, msg )
    else
     return msg
    end
  end %>

<%= "<ul>#{showFamily( self.class, "" )}</ul>" %>
```

In this particular case, the text that is displayed on the web page is slightly different than before since it now shows the class hierarchy of the *view's* class, rather than that of the controller's class. As you will see, views descend from the ActionView::Base class.

You can also divide up a contiguous block of code by placing the individual lines between <% and %> tags instead of placing the entire block between a single pair. The advantage of doing this is that it lets you put standard HTML tags outside the individually delimited lines of Ruby code. You could, for instance, put this into a view:

```
<% arr = ['h','e','l','l','o',' ','w','o','r','l','d'] %>

<% # sort descending from upper value down to nil
reverse_sorted_arr = arr.sort{
     |a,b|
           b.to_s <=> a.to_s
     } %>

<% i = 1 %>
<ul>
<% reverse_sorted_arr.each{ |item| %>
<li><%= "Item [#{i}] = #{item}" %></li>
<% i += 1 %>
<% } %>
</ul>
```

Here, I've assigned an array of chars to the variable, arr, between one set of tags; I've written a block to reverse-sort the array and assigned the result to another variable between a second set of tags; then I've assigned 1 to the variable, i; and then finally I've written this method:

```
reverse_sorted_arr.each{ |item|
  "Item [#{i}] = #{item}"
  i += 1
}
```

But instead of enclosing the method between a single set of tags, I've enclosed each separate line within its own pair of tags. Why should I do this? Well, there are two reasons. First, I want the string in the middle of the block to be displayed on the web page, so I need to use the <%= tag there:

```
<%= "Item [#{i}] = #{item}" %>
```

And secondly, I want the whole set of strings to be displayed as an HTML list. So I've places the <ul> and </ul> tags before and after the Ruby code block; and, I've placed the line of code which displays each array item inside <li> and </li> tags. Notice that these tags are *inside* the Ruby code block, but *outside* the embedded Ruby tags on this particular line:

```
<li><%= "Item [#{i}] = #{item}" %></li>
```

So by dividing up a contiguous block of Ruby code into separately delimited lines, I have been able to do the useful trick of mixing HTML into Ruby code! To be honest, I haven't really mixed it in at all – the Ruby code is still closed off inside the tags; what I've done is told Rails to mix in the HTML at specific points prior to displaying the page in a web browser.

Incidentally, you may find it interesting to compare the version of the application which puts all the embedded Ruby code into the view (**index.html.erb**) with the previous version in which the code was all put into the controller (**say_hello_controller.rb**) and only tiny bits of embedded Ruby (a couple of variables) were placed into the view:

```
<%= @heading %>
<%= @class_hierarchy %>
```

You will probably agree that the first version, in which the code and formatting were kept separate, is neater. On the whole, Ruby code belongs in Ruby code files and HTML formatting belongs in HTML files. While embedded Ruby provides an easy way of letting a view and a controller communicate, it is generally better to keep embedded Ruby code short and simple and put more complex Ruby code into Ruby code files.

LET'S MAKE A BLOG!

For many people, the one thing that really 'turned them on' to Ruby On Rails was the twenty minute demo given by Rails' creator, David Heinemeier Hansson in which he showed how to create a simple weblog. That demo was originally done using Rails 1 and has now been updated (and changed somewhat) for Rails 2. You can watch the Rails 2 demo online here:

http://www.rubyonrails.com/screencasts

A Blog is a great way to show how easy it is to create a fairly complex application using Rails. In the final part of this chapter, I'll explain how you can create a very simple Blog application. I'll use a feature called 'migrations' which will cut out a lot of the hard work of creating the database structure of the 'Model'.

Bear in mind that I have tried to keep the creation of this application as simple as possible and it will not have all the features of a fully functional Blog (there are no user comments and no Administration interface, for example). Once you have completed my basic blog application, you may want to study the screencast tutorial mentioned earlier. This will take you further towards the creation of a more complex blog.

\blog

You may compare the code of your blog application which one which I created. This is supplied in the **\blog** subdirectory of the code accompanying this chapter. This blog application is not 'ready to run', however, as it requires a database which you will have to create. Instead of running it 'as is', use my blog application as a reference to check that the files which you create match those which I created.

Open a command prompt in the directory in which you keep your Rails applications (e.g. **C:\railsapps**) and execute a command to create an application called Blog:

```
rails blog
```

CREATE THE DATABASE

Now let's create a database. Once again, I am assuming that you are using the MySQL database. Open a MySQL prompt (i.e. as before, open the *MySQL Command Line Client* from the MySQL program group). When prompted enter your MySQL password. Now you should see the prompt:

```
mysql>
```

Enter this at the prompt (remember the semicolon at the end):

```
create database blog_development;
```

MySQL should reply 'Query OK' to confirm that this has been created. Now ensure that your database configuration file for your Rails application contains the appropriate entries for the development database. If you are using some other database (not MySQL), your configuration entries must refer to that database.

Open *\app\config\database.yml*. Assuming you are using MySQL, enter '*mysql*' as the adapter, '*localhost*', as the host, your MySQL user name (e.g. '*root*') and your password, if you have one. The database name should match the database which you just created. Here is an example:

```
development:
  adapter: mysql
  host: localhost
  username: root
  database: blog_development
  password: mypassword
```

Remember: If the server is running when you make changes to **database.yml** you should restart the server afterwards!

SCAFFOLDING

We are going to use a feature called 'scaffolding' to create a model, views and controllers all in one go. Scaffolding is a convenient way of getting a simple application up and running quickly. Move into the new **\blog** directory and enter the following at the system prompt:

ruby script/generate scaffold post title:string body:text created_at:datetime

This tells the scaffold generator to create a model comprising Ruby code to access a database table called 'post' with three columns, 'title', 'body' and 'created_at' each of which has the data type (*string*, *text* and *datetime*) specified after the colon.

In order to create the database structure based on this model, we need to run a 'migration' to update the database table itself.

MIGRATION

The scaffold script has created a database migration file for us. Navigate to the **\db\migrate** directory. You will see that this contains a numbered migration file whose name ends with *_create_posts.rb*. If you open this file you can see how the table structure is represented in Ruby code:

```
def self.up
  create_table :posts do |t|
   t.string :title
   t.text :body
   t.datetime :created_at

   t.timestamps
  end
end
```

An application may, over time, gain numerous migrations each of which contains information on a specific iteration of the model - changes and additions made to the table structure of the database. Experienced Rails developers can use migrations selectively to activate different versions of the model. Here, however, we shall use this migration to create the initial structure of the database.

At the system prompt in your application's main directory (e.g. **/blog**), you may use the 'rake' tool to run the migration. Enter this command:

rake db:migrate

After a few moments, you should see a message stating that the rake task has completed and that CreatePosts has been migrated.

PARTIALS

Now let's create a new 'partial' view template. A partial is a fragment of a web page template which Rails may insert, at runtime, into one or more complete web pages.

If, for example, you plan to have the same data entry form in multiple pages on your site, you could create that form inside a partial template. The names of partial templates begin with an underscore.

Create a new file called **_post.html.erb** in your *\app\views\posts\* directory. Open this file and edit its contents to match the following:

```
<div>
<h2><%= link_to post.title, :action => 'show', :id => post %></h2>
<p><%= post.body %></p>
<p><small>
<%= post.created_at.to_s %>
</small></p>
</div>
```

Save your changes. Then open the file named **show.html.erb.** This file was automatically created by the scaffold script. Delete the following 'boilerplate' code from the file...

```
<b>Title:</b>
 <%=h @post.title %>
</p>

<p>
 <b>Body:</b>
 <%=h @post.body %>
</p>

<p>
 <b>Created at:</b>
 <%=h @post.created_at %>
</p>
```

And replace it with this...

```
<%= render :partial => "post", :object => @post %>
```

This tells Rails to render the **_post** partial template at this point. The code in **show.html.erb** should now look like this...

```
<%= render :partial => "post", :object => @post %>

<%= link_to 'Edit', edit_post_path(@post) %> |
<%= link_to 'Back', posts_path %>
```

TEST IT OUT!

And that's it! Now you are ready to test out your application. First, run the server. At the prompt in the **\blog** directory, enter:

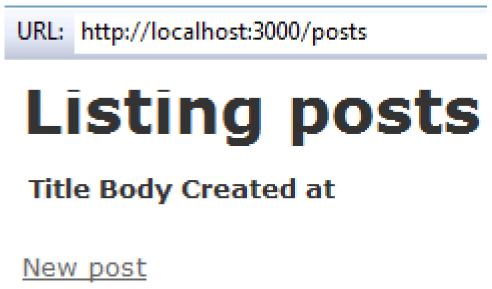
ruby script/server

Recall that if you are not using the default port, 3000, you will need to specify the actual port number after **–p** as explained earlier in this chapter. For example: **ruby script/server –p3003**

Go into your web browser and enter the following address (again, use the actual port number if this is not 3000):

http://localhost:3000/posts

You should see your page with its index page active. This is what should appear...



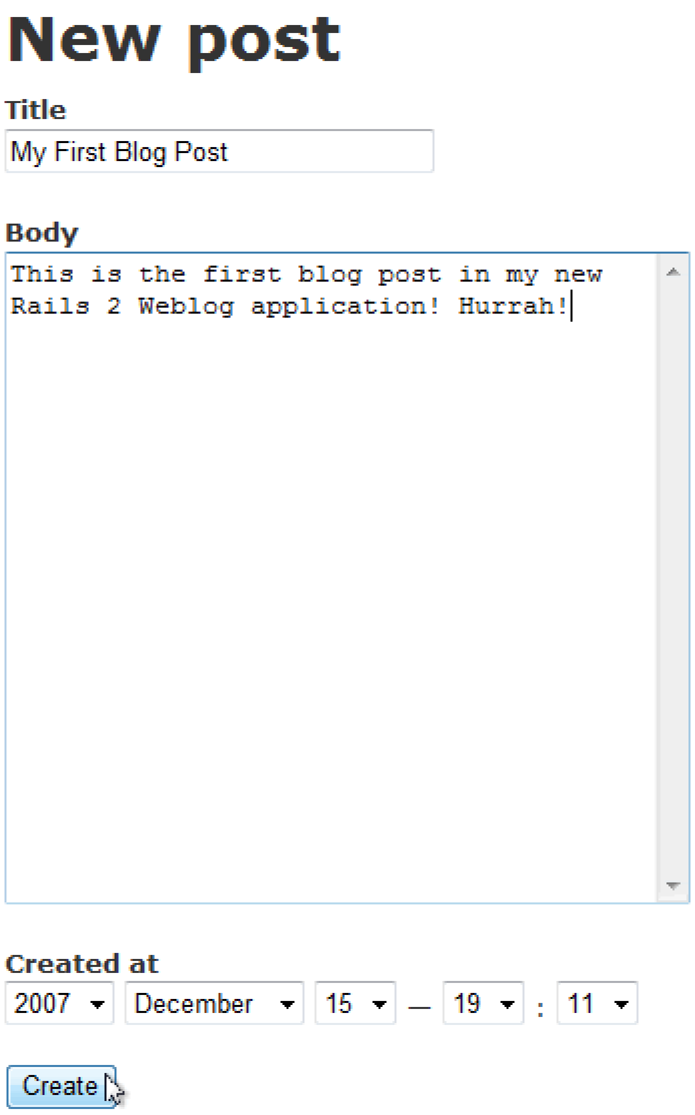
URL: http://localhost:3000/posts

Listing posts

Title Body Created at

New post

Now click the *New Post* link. In the New Post page, enter a title and some body text. Then click *Create*.


New post

Title

My First Blog Post

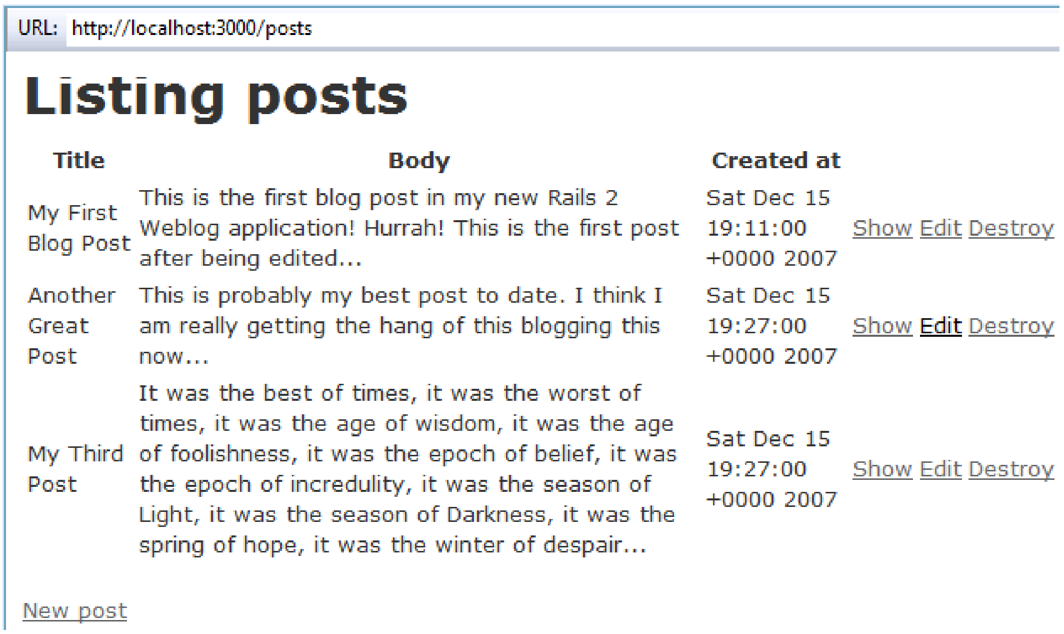
Body

This is the first blog post in my new Rails 2 Weblog application! Hurrah!

Created at

2007 December 15 — 19 : 11

The next page that displays is the ‘Show page’. This is defined by the combination of the **show.html.erb** view and the **_post.html.erb** partial. Now carry on entering posts and clicking the links to navigate through the various defined views...



URL: http://localhost:3000/posts

Listing posts

Title	Body	Created at	
My First Blog Post	This is the first blog post in my new Rails 2 Weblog application! Hurrah! This is the first post after being edited...	Sat Dec 15 19:11:00 +0000 2007	Show Edit Destroy
Another Great Post	This is probably my best post to date. I think I am really getting the hang of this blogging this now...	Sat Dec 15 19:27:00 +0000 2007	Show Edit Destroy
My Third Post	It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair...	Sat Dec 15 19:27:00 +0000 2007	Show Edit Destroy

New post

As mentioned earlier, this chapter assumes that you are using Rails 'in the raw', by entering all the necessary commands at the system prompt. Some IDEs provide a more integrated environment which allows you to generate and code your application using built-in tools and utilities. You will find an overview of some Ruby and Rails IDEs in the Appendix. If you are using Ruby In Steel, you can find an alternative version of this Blog tutorial which employs the integrated tools of Ruby In Steel on the SapphireSteel Software web site:

http://www.sapphiresteel.com/How-To-Create-A-Blog-With-Rails-2

Digging Deeper

MVC

As said earlier, Rails adopts the Model, View and Controller (or MVC) paradigm. While these three component parts are, in theory, separate entities, there is, in practice, inevitably a degree of overlap. For instance, some calculations may be done in the model with others done in the controller; operations that affect the formatting of data could happen in the controller or in the view. There are no hard and fast rules - just a general principle that, as much as possible, operations 'close to the data' should happen in the model, operations 'close to the display' should happen in the view and everything else should go into the controller.

That's MVC in theory. Now let's see how it is implemented by Rails…

MODEL

The Model in Ruby On Rails is a combination of tables in a database – handled by a database server such as MySQL – and a matching set of Ruby classes to work upon those tables. For example, in a Blog you might have a database containing a table called Posts. In that case, the Rails model would also contain a Ruby class named Post (notice that Rails works with plurals – the Posts table can contain many Post objects!). The Ruby Post class would typically contain methods to find, save or load individual Post records from the Posts database. This combination of database tables and a corresponding Ruby classes comprises a Rails Model.

VIEW

The View is pretty much what it sounds like – the visual representation of a Ruby On Rails application. It is (typically bit not necessarily) created in the form of HTML templates with some Ruby code mixed in. In fact, other view types (for example, a graphical view made using Adobe's Flex or Microsoft's Silverlight) are possible but the Rails default is HTML. These templates, which generally have the extension *.html.erb* (but may also use the extension *.rhtml* which was the

Rails 1 default) are not loaded directly into a web browser – after all, web browsers haven't any way of running Ruby code. Instead, they are preprocessed by a separate tool which executes the Ruby code in order to interact with the Model (finding or editing data as necessary), then, as an end result, it creates a new HTML page whose basic layout is defined by an ERb template but whose actual data (that is, the blog posts, shopping cart items or whatever) are provided by the Model. This makes it possible to create highly dynamic web pages which change as a user interacts with them.

CONTROLLER

The Controller takes the form of Ruby code files which act as go-betweens that link the Model and the View. For example, in the web page (View), a user might click a button to add a new post to a blog; using ordinary HTML, this button submits a value named "Create". This causes a method named `create`, in a post 'controller' (a file of Ruby code) to save the new blog entry (some text) which has been entered in the web page (the View) into the database (the data repository of the Model).

THE RAILS FOLDERS

This is a simplified guide to the top-level folders generated by Rails, with a brief description of the files and folders they contain:

app

This contains the code specific to this application. Sub subfolders are: *app\controllers, app\models, app\views* and *app\helpers*.

config

Configuration files for the Rails environment, the routing map, the database, and other dependencies; once a database I defined, includes configuration file, *database.yml*.

db

Contains the database schema in *schema.rb* and may contain code that works on the data in the database.

doc

May contain RDOC documentation (see the Appendix for more on RDOC).

lib

May contain code libraries (that is, code which does not logically belong in *\controllers, \models*, or *\helpers*) for the application.

log

May contain error logs.

public

This directory contains 'static' files which may be used by the web server. It has subdirectories for images, stylesheets, and javascripts.

script

Contains scripts which Rails uses to perform various tasks such as generating certain file types and running the web server.

test

This may contain tests generated by Rails or specified by the user.

tmp

Temporary files used by Rails.

vendor

May contain third-party libraries which do not form a part of the default Rails installation.

OTHER RUBY FRAMEWORKS

Rails may be the most famous Ruby framework but it certainly is not the only one. Others such as Ramaze, Waves and Sinatra, also have a dedicated following. A framework called Merb was once seen as the closest competitor to Rails. However, in December 2008, the Rails and Merb teams announced that they would be collaborating on the next iteration of Rails - 'Rails 3'.

If you are interested in exploring other Ruby frameworks, follow the links below:

Merb: *http://merbivore.com/*

Ramaze: *http://ramaze.net/*

Sinatra: *http://sinatra.rubyforge.org/*

Waves: *http://rubywaves.com/*

The Ramaze developers maintain a more comprehensive list of Ruby frameworks which you can find on their home page.

CHAPTER TWENTY

Dynamic Programming

In the last nineteen chapters, we've covered a huge range of the features of the Ruby language. One thing which we haven't yet looked at in any detail is Ruby's 'dynamic programming' capability.

If you have only used a non-dynamic language (say one of the languages from the C or Pascal family) then dynamism in programming may take a little getting used to. Before going any further, let's clarify what I mean by a 'dynamic' language. The definition is, in fact, a bit vague and not all languages which lay claim to being 'dynamic' share all the same features. In a general sense, however, a language which provides some means by which programs may be modified at runtime can be considered to be dynamic. Another quality of a dynamic language is its ability to change the type of a given variable – something which we have done countless times in the examples throughout this book.

A distinction can be made between a 'dynamically typed' languages such as Ruby and a 'statically typed language' (one in which the type of a variable is predeclared and fixed) such as C, Java or Pascal. In this chapter, I shall be concentrating on the self-modifying capabilities of Ruby.

SELF-MODIFYING PROGRAMS

In most compiled languages and many interpreted languages, writing and running programs are two completely distinct operations. In other words, the code which you write is fixed and beyond any possibility of alteration by the time the program is run.

That is not the case with Ruby. A Ruby program – *its actual code* – can be modified while the program is running. It is even possible to enter new Ruby code at runtime and execute the new code without restarting the program.

The ability to treat data as executable code is called meta-programming. We've been doing meta-programming, albeit of a rather simple sort, throughout this book. Every time you embed an expression inside a double-quoted string you are doing meta-programming. After all, the embedded expression is not really program code – it is a string – and yet Ruby clearly has to 'turn it into' program code in order to be able to evaluate it.

Most of the time you will probably embed rather simple bits of code between the #{ and } delimiters in double-quoted strings. Often you might embed variable names, say, or mathematical expressions:

str_eval.rb

```
aStr = 'hello world'
puts( "#{aStr}" )
puts( "#{2*10}" )
```

But you aren't limited to such simple expressions. You could, if you wished, embed just about anything into a double-quoted string. You could, in fact, write an entire program in a string. You don't even need to display the end result using print or puts. Just placing a double-quoted string into your program will cause Ruby to evaluate it:

```
"#{def x(s)
     puts(s.reverse)
  end;
(1..3).each{x(aStr)}}"
```

Writing a whole program inside a string would probably be a pretty pointless endeavour. However, there are other occasions when this, and similar, features can be used much more productively. For example, the Rails framework makes extensive use of meta-programming. You may use meta-programming to explore artificial intelligence and 'machine learning'. In fact, any application which would benefit from having a program's behaviour modified due to interaction during the program's execution is a prime candidate for meta-programming.

Dynamic (meta-programming) features are ubiquitous in Ruby. Consider, for example, attribute accessors: passing a symbol (such as :aValue) to the attr_accessor method causes two methods (aValue and aValue=) to be created.

EVAL MAGIC

The eval method provides a simple way of evaluation a Ruby expression in a string. At first sight, eval may appear to do exactly the same job as the #{ } markers delimiting expressions in a double-quoted string. The two lines of code following produce identical results:

eval.rb

```
puts( eval("1 + 2" ) )
puts( "#{1 + 2}" )
```

There are times, however, when the results may not be what you are expecting. Look at the following, for instance:

eval_string.rb

```
exp = gets().chomp()
puts( eval( exp ))
puts( "#{exp}" )
```

Let's suppose that you enter 2 * 4 and this is assigned to exp. When you evaluate exp with eval the result is 8 but when you evaluate exp in a double-quoted string, the result is '**2*4**'. This is because anything read in by gets() is a string and "#{exp}" evaluates it *as a string* and not as an expression, whereas eval(exp) evaluates a string *as an expression*.

In order to force evaluation inside a string, you could place eval in the string (though that, admittedly, might defeat the object of the exercise):

```
puts( "#{eval(exp)}" )
```

Here is another example. Try it out and follow the instructions when prompted:

eval2.rb

```
print( "Enter the name of a string method (e.g. reverse or upcase): " )
                                                    # user enters: upcase
methodname = gets().chomp()
exp2 = "'Hello world'."<< methodname
puts( eval( exp2 ) )                                #=> HELLO WORLD
puts( "#{exp2}" )                                   #=> 'Hello world'.upcase
puts( "#{eval(exp2)}" )                             #=> HELLO WORLD
```

The **eval** method can evaluate strings spanning many lines, making it possible to execute an entire program embedded in a string:

eval3.rb

```
eval( 'def aMethod( x )
  return( x * 2 )
end

num = 100
puts( "This is the result of the calculation:" )
puts( aMethod( num ))' )
```

With all this **eval** cleverness, let's now see how easy it is to write a program which can itself write programs. Here it is:

eval4.rb

```
input = ""
until input == "q"
  input = gets().chomp()
  if input != "q" then eval( input ) end
end
```

This may not look much and yet this little a program lets you create and execute real workable Ruby code from a prompt. Try it out. Run the program and enter these two methods one line at a time (but *don't hit 'q' to quit yet* - we'll be writing some more code in a moment):

```
def x(aStr); puts(aStr.upcase);end
def y(aStr); puts(aStr.reverse);end
```

Note that you have to enter each whole method on a single line since my program evaluates every line as it is entered. I'll explain how to get around that limitation later on. Thanks to eval each method is turned into real, workable Ruby code. You can prove this by entering the following:

```
x("hello world")
y("hello world")
```

Now, these expressions are themselves evaluated and they will call the two methods we wrote a moment ago, resulting in this output:

```
HELLO WORLD
dlrow olleh
```

Not bad for just five lines of code!

SPECIAL TYPES OF EVAL

There are some variations on the eval theme in the form of the methods named instance_eval, module_eval and class_eval. The instance_eval method can be called from a specific object and it provides access to the instance variables of that object. It can be called either with a block or with a string:

instance_eval.rb

```
class MyClass
 def initialize
   @aVar = "Hello world"
 end
end

ob = MyClass.new
p( ob.instance_eval { @aVar } )          #=> "Hello world"
p( ob.instance_eval( "@aVar" ) )         #=> "Hello world"
```

The eval method, on the other hand, cannot be called from an object in this way due to the fact that it is a private method of Object (whereas instance_eval is public). In fact, you could explicitly change the visibility of eval by sending its name (the symbol :eval) to the public method, though gratuitous messing about with method visibility in the base class is not, in general, recommended!

Strictly speaking, eval is a method of the Kernel module which is mixed into the Object class. In fact, it is the Kernel module which provides most of the functions available as methods of Object.

You can change the visibility of eval by adding to the definition of the Object class in this way:

```
class Object
  public :eval
end
```

Indeed, bearing in mind that, when you are writing 'free-standing' code you are actually working within the scope of Object, simply entering this code (without the class Object 'wrapper') would have the same effect:

```
public :eval
```

Now you can use eval as a method of the ob variable:

```
p( ob.eval( "@aVar" ) )            #=> "Hello world"
```

The module_eval and class_eval methods operate on modules and classes rather than on objects. For example, this code adds the xyz method to the X module (here xyz is defined in a block and added as an instance method of the receiver by define_method which is a method of the Module class); and it adds the abc method to the Y class:

module_eval.rb

```
module X
end

class Y
  @@x = 10
  include X
end

X::module_eval{ define_method(:xyz){ puts("hello" ) } }
Y::class_eval{ define_method(:abc){ puts("hello, hello" ) } }
```

When accessing class and module methods you may use the scope resolution operator :: or a single dot. The scope resolution operator is obligatory when accessing constants and optional when accessing methods.

So, now an object that is an instance of Y will have access to both the abc method of the Y class and the xyz method of the X module which has been mixed into the Y class:

```
ob = Y.new
ob.xyz              #=> "hello"
ob.abc              #=> "hello, hello"
```

In spite of their names, module_eval and class_eval are functionally identical and each may be used with ether a module or a class:

```
X::class_eval{ define_method(:xyz2){ puts("hello again" ) } }
Y::module_eval{ define_method(:abc2){ puts("hello, hello again" ) } }
```

You can also add methods into Ruby's standard classes in the same way:

```
String::class_eval{ define_method(:bye){ puts("goodbye" ) } }
"Hello".bye         #=> "goodbye"
```

ADDING VARIABLES AND METHODS

The module_eval and class_eval methods can also be used to retrieve the values of class variables (but bear in mind that the more you do this, the more your code becomes dependent on the implementation details of a class, thereby breaking encapsulation):

```
Y.class_eval( "@@x" )
```

In fact, class_eval can evaluate expressions of arbitrary complexity. You could, for example, use it to add new methods to a class by evaluating a string...

```
ob = X.new
X.class_eval( 'def hi;puts("hello");end' )
ob.hi                    #=> "hello"
```

Returning to the earlier example of adding and retrieving class variables from *outside* a class (using class_eval); it turns out that there are also methods designed to do this from *inside* a class. The methods are called class_variable_get (this takes a symbol argument representing the variable name and it returns the variable's value) and class_variable_set (this takes a symbol argument representing a variable name and a second argument which is the value to be assigned to the variable). nHere is an example of these methods in use:

classvar_getset.rb

```
class X
  def self.addvar( aSymbol, aValue )
    class_variable_set( aSymbol, aValue )
  end

  def self.getvar( aSymbol )
    return class_variable_get( aSymbol )
  end
end

X.addvar( :@@newvar, 2000 )
puts( X.getvar( :@@newvar ) )   #=> 2000
```

To obtain a list of class variable names as an array of strings, use the class_variables method:

```
p( X.class_variables )      #=> ["@@abc", "@@newvar"]
```

You can also add instance variables to classes and objects after they have been created using instance_variable_set this:

```
ob = X.new
ob.instance_variable_set("@aname", "Bert")
```

Combine this with the ability to add methods and the bold (or maybe reckless?) programmer can completely alter the internals of a class 'from the outside'. Here I have implemented this in the form of a method called addMethod in class X, which uses the send method to create a new method, m, using define_method with the method body, defined by &block:

dynamic.rb

```
def addMethod( m, &block )
  self.class.send( :define_method, m , &block )
end
```

The send method invokes the method identified by the first argument (a symbol), passing to it any arguments specified.

Now, an X object can call addMethod to insert a new method into the X class:

```
ob.addMethod( :xyz ) { puts("My name is #{@aname}") }
```

Although this method is called from a specific instance of the class (here ob) it affects the class itself so the newly defined method will also be available to any subsequent instances (here ob2) created from the X class:

```
ob2 = X.new
ob2.instance_variable_set("@aname", "Mary")
ob2.xyz
```

If you don't care about the encapsulation of data in your objects you can also retrieve the value of instance variables using the instance_variable_get method:

```
ob2.instance_variable_get( :@aname )
```

You can similarly set and get constants:

```
X::const_set( :NUM, 500 )
puts( X::const_get( :NUM ) )
```

As const_get returns the value of a constant, you could use this method to get the value of a class name and then append the new method to create a new object from that class. This could even give you a way of creating objects at runtime by prompting the user to enter class names and method names. Try this out by running this program:

dynamic2.rb

```
class X
  def y
    puts( "ymethod" )
  end
end

print( "Enter a class name: ")                 #<= Enter: X
cname = gets().chomp
ob = Object.const_get(cname).new
p( ob )
print( "Enter a method to be called: " )       #<= Enter: y
mname = gets().chomp
ob.method(mname).call
```

CREATING CLASSES AT RUNTIME

So far we have modified classes and created new objects from existing classes. But how would you go about creating a completely new class at runtime? Well, just as const_get can be used to access an existing class, const_set can be used to create a new class. Here's an example of how to prompt the user for the name of a new class before creating that class, adding a method (myname) to it, creating an instance, x, of that class and calling its myname method:

create_class.rb

```
puts("What shall we call this class? ")
className = gets.strip().capitalize()
Object.const_set(className,Class.new)
puts("I'll give it a method called 'myname'" )
className = Object.const_get(className)
className::module_eval{ define_method(:myname){
  puts("The name of my class is '#{self.class}'" ) }
  }
x = className.new
x.myname
```

BINDINGS

The eval method may take an optional 'binding' argument which, if provided, causes the evaluation to be done within a specific scope or 'context'. It probably won't come as any surprise to discover that, in Ruby, a binding is an instance of the Binding class. You can return a binding using the binding method. The documentation of eval in the Ruby class library provides this example:

binding.rb

```
def getBinding(str)
  return binding()
end
str = "hello"
puts( eval( "str + ' Fred'" )   )                  #=> "hello Fred"
puts( eval( "str + ' Fred'", getBinding("bye") ) )  #=> "bye Fred"
```

Here binding is a private method of Kernel. The getBinding method is able to call binding within the current context and return the current value of str. At the time of the first call to eval the context is the *main* object and the value of the local variable, str, is used; in the second call, the context moves inside the getBinding method and the local value of str is now that of the str argument to the method.

The context may also be defined by a class. In **binding2.rb** you can see that the values of the instance variable @mystr and the class variable @@x vary according to the class.:

binding2.rb

```
class MyClass
  @@x = " x"
  def initialize(s)
    @mystr = s
  end
  def getBinding
    return binding()
  end
end

class MyOtherClass
  @@x = " y"
  def initialize(s)
    @mystr = s
  end
  def getBinding
    return binding()
  end
end

@mystr = self.inspect
@@x = " some other value"

ob1 = MyClass.new("ob1 string")
ob2 = MyClass.new("ob2 string")
ob3 = MyOtherClass.new("ob3 string")
```

```
puts(eval("@mystr << @@x", ob1.getBinding))  #=> ob1 string x
puts(eval("@mystr << @@x", ob2.getBinding))  #=> ob2 string x
puts(eval("@mystr << @@x", ob3.getBinding))  #=> ob3 string y
puts(eval("@mystr << @@x", binding))         #=> main some other value
```

SEND

You may use the **send** method to call a method with the same name as the specified symbol:

send1.rb

```
name = "Fred"
puts( name.send( :reverse ) )     #=> derF
puts( name.send( :upcase ) )      #=> FRED
```

Although the **send** method is documented as requiring a symbol argument, you may also use a string argument. Or, for consistency, you could use **to_sym** to transform and then call the method with the same name:

```
name = MyString.new( gets() )
methodname = gets().chomp.to_sym #<= to_sym is not strictly necessary
name.send(methodname)
```

Here is a working example of using **send** to execute a named method entered at runtime:

send2.rb

```
class MyString < String
  def initialize( aStr )
    super aStr
  end

  def show
    puts self
  end
```

```
    def rev
      puts self.reverse
    end
  end

  print("Enter your name: ")            #<= Enter: Fred
  name = MyString.new( gets() )
  print("Enter a method name: " )       #<= Enter: rev
  methodname = gets().chomp.to_sym
  puts( name.send(methodname) )         #=> derF
```

Recall how we used `send` earlier (**dynamic.rb**) to create a new method by calling `define_method` and passing to it the name, `m`, of the method to be created and a block, `&block`, containing the code of the new method:

dynamic.rb

```
  def addMethod( m, &block )
    self.class.send( :define_method, m , &block )
  end
```

REMOVE METHODS

In addition to creating new methods, there may be times when you want to remove existing methods. You can do this using `remove_method` within the scope of a given class. This removes the method specified by a symbol from a specific class:

rem_methods1.rb

```
  puts( "hello".reverse )
  class String
    remove_method( :reverse )
  end
  puts( "hello".reverse )     #=> 'undefined method' error!
```

If a method with the same name is defined for an ancestor of that class, the ancestor class method is *not* removed:

rem_methods2.rb

```
class Y
  def somemethod
    puts("Y's somemethod")
  end
end

class Z < Y
  def somemethod
    puts("Z's somemethod")
  end
end

zob = Z.new
zob.somemethod                          #=> "Z's somemethod"

class Z
  remove_method( :somemethod )
end

zob.somemethod                          #=> "Y's somemethod"
```

The `undef_method`, by contrast, prevents the specified class from responding to a method call even if a method with the same name is defined in one of its ancestors:

undef_methods.rb

```
zob = Z.new
zob.somemethod                          #=> "Z's somemethod"

class Z
  undef_method( :somemethod )
end

zob.somemethod                          #=> 'undefined method' error
```

HANDLING MISSING METHODS

When Ruby tries to execute an undefined method (or, in OOP terms, when an object is sent a message which it cannot handle), the error causes the program to exit. You may prefer your program to recover from such an error. You can do this by writing a method named method_missing, with an argument to which the missing method's name is assigned. This will execute when a non-existent method is called:

nomethod1.rb

```
def method_missing( methodname )
  puts( "#{methodname} does not exist" )
end
xxx                   #=> displays: 'xxx does not exist'
```

The method_missing method can also take a list of incoming arguments (*args) after the missing method name:

nomethod2.rb

```
def method_missing( methodname, *args )
    puts( "Class #{self.class} does not understand:
                  #{methodname}( #{args.inspect} )" )
end
```

The method_missing method could even create the undefined method dynamically:

```
def method_missing( methodname, *args )
     self.class.send( :define_method, methodname,
            lambda{ |*args| puts( args.inspect) } )
end
```

WRITING PROGRAMS AT RUNTIME

Finally, let's go back to the program we looked at earlier (**eval4.rb**) which prompted the user to enter strings to define code at runtime, evaluated those strings and created new runnable methods from them.

One drawback of that program was that it insisted that each method was entered on a single line. It is, in fact, pretty simple to write a program that allows the user to enter methods spanning many lines. Here, for example, is a program which evaluates all the code entered up until a blank line is entered:

writeprog.rb

```
program = ""
input = ""
line = ""
until line.strip() == "q"
  print( "?- " )
  line = gets()
  case( line.strip() )
  when ''
    puts( "Evaluating..." )
    eval( input )
    program += input
    input = ""
  when 'l'
    puts( "Program Listing..." )
    puts( program )
  else
    input += line
  end
end
```

You can try this out by entering whole methods followed by blank lines, like this (just enter the code, of course, not the comments):

```
def a(s)              # <= press Enter after each line
return s.reverse      # <= press enter (and so on...)
end
     # <- Enter a blank line here to eval these two methods
def b(s)
return a(s).upcase
end
     # <- Enter a blank line here to eval these two methods
puts( a("hello" ) )
     # <- Enter a blank line to eval
     #=> Displays "olleh"
puts( b("goodbye" ) )
     # <- Enter a blank line to eval
     #=> Displays "EYBDOOG"
```

After each line entered a prompt ('?-') appears except when the program is in the process of evaluating code, in which case, it displays '*Evaluating*' or when it shows the result of an evaluation, such as "*olleh*".

If you enter the text exactly as indicated above, this is what you should see:

```
Write a program interactively.
Enter a blank line to evaluate.
Enter 'q' to quit.
?- def a(s)
?- return s.reverse
?- end
?-
Evaluating...
?- def b(s)
?- return a(s).upcase
?- end
?-
Evaluating...
?- puts(a("hello"))
?-
```

```
Evaluating...
olleh
?- b("goodbye")
?-
Evaluating...
EYBDOOG
```

This program is still very simple. It doesn't have any basic error recovery let alone fancy stuff such as file saving and loading. Even so, this small example demonstrates just how easy it is to write self-modifying programs in Ruby. Using the techniques outlined in this chapter, you could create anything from a natural language parser that can be taught rules of grammar to an adventure game which can learn new puzzles.

In this book we've covered a lot of ground – from "hello world" to dynamic programming. The rest is up to you.

This is where the adventure really begins.

Digging Deeper

FREEZING OBJECTS

With all these ways of modifying objects at your disposal, you may be concerned that objects are at risk of being modified unintentionally. In fact, you can specifically fix the state of an object by 'freezing' it (using the `freeze` method). Once frozen, the data contained by an object cannot be modified and, if an attempt is made to do so, a TypeError exception will be raised. Take care when freezing an object, however, as, once frozen, it cannot be 'unfrozen'.

freeze.rb

```
s = "Hello"
s << " world"
s.freeze
s << " !!!"    # Error: "can't modify frozen string (TypeError)"
```

You can specifically check if an object is frozen using the `frozen?` method:

```
a = [1,2,3]
a.freeze
if !(a.frozen?) then
  a << [4,5,6]
end
```

Be aware that, while the data of a frozen object cannot be modified, the class from which it is defined can be modified. Let's suppose that you have a class X which contains a method, `addMethod`, which can create new methods with the name given by a symbol, `m`:

cant_freeze.rb

```
def addMethod( m, &block )
  self.class.send( :define_method, m , &block )
end
```

Now, if you have an object, `ob`, created from the M class, then it is perfectly legitimate to call `addMethod`, to add a new method to class M:

```
ob.freeze
ob.addMethod( :abc ) { puts("This is the abc method") }
```

If you want to prevent a frozen object from modifying its class, you could, of course, test its state using the `frozen?` method:

```
if not( ob.frozen? ) then
  ob.addMethod( :def ) { puts("'def' is not a good name for a method") }
end
```

You can also freeze the class itself (remember, a class is also an object):

```
X.freeze
if not( X.frozen? ) then
  ob.addMethod( :def ) { puts("'def' is not a good name for a method") }
end
```

APPENDICES

Appendix A: Documenting Ruby With RDOC

RDoc describes a source code documentation format and tool. The RDoc tool, which comes as standard with Ruby, can process Ruby code files and the C-code Ruby class library in order to extract documentation and format it so that it can be displayed in, for example, a web browser. RDoc documentation can be explicitly added in the form of source code comments. The RDoc tool can also extract elements of the source code itself to provide the names of classes, modules and methods along with the names of any arguments required by methods.

It is easy to document your own code in a way that is accessible to the RDoc processor. You can either write a block of ordinary single-line comments before the code being documented (such as a class or method) or you can write an embedded multi-line comment delimited by `=begin rdoc` and `=end`. Note that `rdoc` must follow `=begin`, otherwise the RDoc processor will ignore the comment block.

When you want to generate the documentation, you just need to run the RDoc processor from the command prompt. To generate documentation for a single file, enter **rdoc** followed by the name of the file:

```
rdoc rdoc1.rb
```

To generate documentation for multiple files, enter the file names separated by spaces after the **rdoc** command:

```
rdoc rdoc1.rb rdoc2.rb rdoc3.rb
```

The Rdoc tool will create a nicely formatted HTML file (*index.html*) with three panes at the top and a fourth, larger pane at the bottom. The three top panes display the names of the files, classes and methods while the bottom pane displays the documentation.

The HTML contains hyperlinks so that you can click class and method names to navigate to the associated documentation. The documentation is placed into its own subdirectory, *\doc*, along with a number of required HTML files and a style sheet to apply formatting.

You can add extra formatting to your RDoc comments by placing formatting characters around single words or tags around multiple words. Use *and* for bold, _and_ for italic and +and+ for a 'typewriter' font. The equivalent tags for longer pieces of text are <b>and</b> for bold, <em>and</em> for italic and <tt>and</tt> for typewriter.

If you want to exclude comments, or parts of a comment, from the RDoc documentation, you can place it between #-- and #++ comment markers, like this:

```
#--
# This comment won't appear
# in the documentation
#++
# But this one will
```

There are also various special instructions available, enclosed between pairs of colons. For instance, if you want to add a title to be displayed in the browser bar, use :title: like this:

```
#:title: My Fabulous RDoc Document
```

There are many more options available with RDoc to enable you to format documentation in a variety of ways and output in alternative formats to HTML. If you really want to master RDoc, be sure to read the complete documentation:

http://rdoc.sourceforge.net/doc/index.html

Appendix B: Installing MySQL For Ruby On Rails

If you are working with Rails, you will need to install a database. While there are quite a few possible choices available to you, one of the most widely used is MySQL. If you've never used MySQL before, you may find some of the setup options confusing. Here, I'll try to guide you through the process to avoid potential problems.

This guide is based on an installation of MySQL 5.0 under Windows. There may be differences when installing other versions on other operating systems. Refer to the MySQL site for additional guidance.

The MySQL main site is at **http://www.mysql.com/** and from here you can navigate to the download page for the current version.

DOWNLOAD MYSQL

I shall assume that you will be using the free edition of MySQL. This is available for download from **http://dev.mysql.com/downloads**. The current version, at the time of writing, is MySQL 5 Community Server. The name and version number will, of course, change over time. Download whichever is the current (not upcoming, alpha or beta) release. Choose the specific version recommended for your operating system (there may be different versions for Win32 and Win64, for example).

You will need to scroll some way down this page to locate the installer for your operating system. For Windows, you can either download the complete MySQL package or the smaller Windows Essentials package. The complete package contains extra tools for database developers but these are not required for simple Rails development. For most people, therefore, the smaller Windows Essentials download file is the one to get.

You should click the 'Pick A Mirror' link alongside this option. You will then be shown a questionnaire which you can fill out if you wish. If you don't wish to do so, just scroll down the page and pick a regional download site. Click a link and save the file, which will be named something like (the numbers may differ): **mysql-essential-5.0.41-win32.msi**, to any convenient directory on your disk.

INSTALL MYSQL

Once the download has completed run the program by selecting *Open* or *Run* in the download dialog if this is still visible, or by double-clicking the installation file via the Windows Explorer.

Note: During the installation of MySQL some advertising screens may appear. Click the buttons to move through these. Some security warnings may also prompt you to verify your intention to install the software. When prompted, you should click the necessary options to continue with the installation.

The first page of the Setup Wizard will now appear. Click the *Next* button.

You can either leave the *Typical* setup option selected if you are happy to install the software into the default MySQL directory beneath **C:\Program Files**. If you want to install to some other directory, however, select *Custom*. Then click *Next*. Click *Change* to change the directory.

When you are ready to move on, click *Next*.

You will see the screen stating 'Ready To Install the Program'. Verify that the destination folder is correct, then click the *Install* button.

Depending on the version of MySQL you may now either be shown some promotional screens or you may be prompted to create a new MySQL account which will let you receive news of changes and updates. These are not an essential part of the software installation and you may click the *Next* or *Skip* buttons to move on through the installation.

The Wizard Completed dialog now appears.

Click the *Finish* button.

CONFIGURE MYSQL

In fact, this isn't the end of the installation after all. With some installers, a new screen pops up now welcoming you to the MySQL Server Instance Configuration Wizard. If this does not occur, you will need to load this yourself. On Windows, click the Start menu, then navigate through your program groups to MySQL->MySQL Server 5.0 (or whichever version number you are using) then MySQL Server Instance Config Wizard. Click *Next*.

Assuming that this is the first time you've installed MySQL on this machine, you can select Standard Configuration (if you are upgrading from an older version of MySQL you need to select Detailed Configuration – that is beyond the scope of this simple setup guide). Click *Next*. In the next dialog, leave the default options selected (i.e. *Install As Windows Service; Service Name = 'MySQL'* and *Launch the MySQL Server automatically*). Then click *Next*.

In the next screen, leave 'Modify Security Settings' checked and enter the same password (of your choice) into the first two text fields. You will need this password later so remember it or write it down in a secure location. If you may need to access MySQL from another computer you can check 'Enable root access from remote machines'. Then click *Next*.

Note: The default MySQL user name is 'root'. The password is the one you just entered. You will need both these items of information later when creating Rails applications.

The next screen just gives you some information about the tasks that are about to be performed. Click the *Execute* button.

If you have previously installed or configured MySQL, you may see an error message which tells you to Skip the installation. You may click *Retry* to see if you can bypass this problem. If not, press *Skip* and then restart the MySQL configuration process, selecting *Reconfigure Instance* and Standard Instance when prompted.

When everything is installed this screen appears. Click *Finish*.

And that's it!

Appendix C: Further Reading

Here is a short list of some of the most useful books on Ruby and Rails...

BOOKS

Beginning Ruby: From Novice To Professional
by Peter Cooper $39.99
APress: http://www.apress.com
ISBN: 1590597664
The book is well written, nicely laid out, the explanations are clear and the code examples are useful. In short, if you already have some programming experience and want an accessible introduction to the world of Ruby, this is the book to get.

Programming Ruby: The Pragmatic Programmer's Guide
by Dave Thomas, with Chad Fowler and Andy Hunt $44.95
ISBN: 0-9745140-5-5 (2nd edition)
ISBN: 9781934356081 (3rd edition)
Pragmatic: http://www.pragmaticprogrammer.com/titles/ruby/index.html
A vast (more than 860 pages) guide to the Ruby language and libraries, the so-called 'pickaxe book' is generally considered to be an essential Ruby reference. Not a light read, though, and not (in my view) the best 'first book' on Ruby. All the same, you may need it sooner or later. The 2nd edition covers Ruby 1.8; the 3rd edition covers Ruby 1.9.

The Ruby Way
by Hal Fulton $39.99
Addison-Wesley: http://www.awprofessional.com/ruby
ISBN: 0-672-32884-4
In the introductory section, the author states that, due to its relative lack of 'tutorial' material, "You probably won't learn Ruby from this book." He describes it more as a "sort of 'inverted reference'. Rather than looking up the name of a method or a class, you will look things up by function or purpose." Personally, I think he greatly underestimates the tutorial value of The Ruby Way. The author doers, however, assume that long as you are reasonably adept at programming.

Ruby On Rails Up and Running
by Bruce A. Tate & Curt Hibbs $29.99
O'Reilly: www.oreilly.com
ISBN: 0-596-10132-5
I prefer programming books to get right down to business without too much in the way of waffle. I frankly don't have the patience to wade through 1000+ page tomes or follow step-by-step guides to building monolithic applications. So this book appeals to me. In just seven chapters, it covers all the really important stuff about Rails – its design patterns and classes; its scripts and application-generation tools; its Models, Views, Controllers and Scaffolding; plus an overview of using Ajax and Unit Testing.

Ruby For Rails
by David A. Black $44.95
Manning : www.manning.com/black
ISBN 1-932394-69-9
While this book concentrates on Rails development, at every step of the way it delves into the inner workings of the underlying Ruby code. In seventeen chapters and just under 500 pages, it takes you from a first look at the Ruby language through to the specifics of creating Rails models, controllers, views, helpers and templates. Along the way it explains a great deal about Ruby including its arrays and hashes, classes, methods, blocks and iterators. In short, if you are new to Ruby but want to get up to speed on Rails as rapidly as possible, this book could be what you are looking for.

Agile Web Development With Rails (3rd edition)
by Sam Ruby, Dave Thomas and David Heinemeier Hansson $43.95
Pragmatic: http://pragprog.com/titles/rails3/agile-web-development-with-rails-third-edition
ISBN: 9781934356166
This is the 'must have' book on Rails. There are several Ruby programming books which might compete for the claim to being 'essential', but I know of no other Rails book that comes anywhere close to rivalling Agile Web Development for its comprehensive coverage of its subject. 'Nuff said: if you are serious about Ruby On Rails, buy this book! The 3rd edition covers Rails 2.

E-Books

Learn To Program
The first edition of Chris Pine's book provides a gentle introduction to Ruby.
http://pine.fm/LearnToProgram/

Programming Ruby: The Pragmatic Programmer's Guide
The first edition of the well-known 'pickaxe book'.
http://www.rubycentral.com/book/

Why's Poignant Guide To Ruby
Possibly the strangest programming book you'll ever read - complete with talking foxes!
http://poignantguide.net/ruby/

The Little Book Of Ruby
The baby brother to the book you are currently reading.
http://www.sapphiresteel.com/The-Little-Book-Of-Ruby

Appendix D: Web Sites

There are innumerable web sites devoted to Ruby, Rails and related technologies. Here are a few to start exploring...

RUBY AND RAILS INFORMATION

Ruby language site
http://www.ruby-lang.org

Ruby documentation site
http://www.ruby-doc.org/

Ruby class library reference (online)
http://www.ruby-doc.org/core/

Ruby class library reference (to download)
http://www.ruby-doc.org/downloads

Ruby On Rails Site
http://www.rubyonrails.org/

The blog of the author of The Book Of Ruby...
http://www.sapphiresteel.com/-Blog-

Appendix E: Ruby and Rails Development Software

IDES/EDITORS

3rd Rail
http://www.codegear.com/products/3rdrail/
Commercial Rails-centric IDE for Eclipse.

Aptana IDE/ RADRails
http://www.aptana.com/
Free Rails-centric IDE for Eclipse.

Komodo
http://www.activestate.com/
Multi-language (Ruby, Python, PHP, Perl, Pcl), cross-platform commercial IDE. Free version also available.

NetBeans
http://www.netbeans.org/products/ruby/
Free Ruby IDE for NetBeans.

Ruby In Steel
http://www.sapphiresteel.com/
Commercial Ruby and Rails IDE for Visual Studio. Free version also available.

TextMate
http://www.macromates.com/
Ruby editor for Mac OS X

WEB SERVERS

Here are some popuar web servers for use with Ruby On Rails

WEBrick
http://www.webrick.org/

LightTPD
http://www.lighttpd.net/

Mongrel
http://mongrel.rubyforge.org/

Apache
http://www.apache.org/

DATABASES

MySQL
http://www.mysql.com/

SQLite
http://www.sqlite.org/

PostgreSQL
http://www.postgresql.org/

SQL Server Express
http://www.microsoft.com/sql/editions/express/

Appendix F: Ruby Implementations

At the time of writing, versions of both Ruby 1.8.x and 1.9.1 are available and version 2.0 is promised for some future date. Currently Ruby 1.8.6 is probably the most widely used version of Ruby and most of this book applies to Ruby 1.8. In fact, in spite of the release of Ruby 1.9, Ruby 1.8.x will remain an important Ruby platform for some time in order to support projects developed using that version, including applications built using Ruby On Rails and other frameworks. Other Ruby interpreters, compilers and virtual machines are also available or in development. Here is a short list of web sites that will provide more information on (and downloads of) implementations of Ruby...

Ruby
The 'standard' Ruby implementation
http://www.ruby-lang.org/en/downloads/

JRuby
Ruby For Java
http://www.headius.com/

Iron Ruby
Microsoft's in-development 'Ruby For .NET'
http://www.ironruby.net/

Rubinius
Compiler/Virtual Machine for Ruby (written largely in Ruby)
http://rubini.us/

Maglev
Fast Ruby implementation (in development)
http://maglev.gemstone.com/

INDEX