the little book of... ruby

BY

HUW COLLINGBOURNE

4th Edition

YOUR FREE GUIDE TO PROGRAMMING RUBY

FROM BITWISE COURSES

WWW.BITWISECOURSES.COM

AND

SAPPHIRESTEEL SOFTWARE

WWW.SAPPHIRESTEEL.COM

The Little Book Of Ruby

written by
Huw Collingbourne

Download source code from:
http://www.sapphiresteel.com/ruby-programming/The-Little-Book-Of-Ruby.html

First edition: June 2006
Second edition: March 2008
Third edition: July 2011
Fourth edition: July 2015

TABLE OF CONTENTS

Table of Contents 3
Welcome To The Little Book Of Ruby 6
Learn Ruby In Ten Chapters… 6
What Is Ruby? 7
What Is Rails? 7
Installing Ruby 8
Ruby 1 or Ruby 2? 8
Installing A Ruby Editor or IDE 9
SciTE 9
Komodo Edit 9
RadRails 9
RubyMine 10
TextMate 10
Sapphire 10
Get The Source Code Of The Sample Programs 11
Running Ruby Programs 11
Running Ruby From a Command Prompt 12
On Windows...12
On Mac OS X 12
How To Use This Book 14
Making Sense Of The Text 14
Chapter One..15
Strings and Embedded Evaluation 18
Methods 19
Numbers 21

Testing a Condition: if … then....22

Chapter Two....24

Instances and Instance Variables....26

Constructors – new and initialize....28

Inspecting Objects....31

Chapter Three....35

Creating a Class Hierarchy....35

One Parent, Many Children…....37

Superclasses and Subclasses....39

Chapter Four....41

Accessor Methods....41

Attribute Readers and Writers....43

Attributes Create Variables....46

Calling Methods of a Superclass....48

Class Variables....49

Objects 'share' Class variables....51

Chapter Five....52

Introducing Arrays....52

Creating Arrays....53

Multi-Dimensional Arrays....56

Iterating Over Arrays....57

Indexing Into Arrays....59

Chapter Six....62

Creating Hashes....62

Indexing Into A Hash....65

Hash Operations....66

Chapter Seven....68

for Loops....68

Blocks....72

while Loops 73
While Modifiers 74
Until Loops 76
Chapter Eight 78
if..else 79
and..or..not 81
if..elsif 82
? : 83
unless 84
if and unless Modifiers 84
Case Statements 86
Chapter Nine 89
A Module Is Like A Class… 89
Module Methods 90
Modules as Namespaces 91
Module 'Instance Methods' 93
Included Modules or 'Mixins' 93
Including Modules From Files 97
Pre-Defined Modules 98
Chapter Ten 99
Saving Data 99
YAML 99
Files 100
Moving On 101

Welcome To The Little Book Of Ruby

Learn Ruby In Ten Chapters…

Chapter One :	Strings and Methods
Chapter Two:	Classes and Objects
Chapter Three:	Class Hierarchies
Chapter Four:	Accessors, Attributes, Class Variables
Chapter Five:	Arrays
Chapter Six:	Hashes
Chapter Seven:	Loops and Iterators
Chapter Eight:	Conditional Statements
Chapter Nine:	Modules and Mixins
Chapter Ten:	Saving Files, Moving On…

What Is Ruby?

Ruby is a cross-platform, object-oriented, interpreted language which has many features in common with other 'scripting' languages such as Perl and Python as well as with 'pure' object-oriented languages such as Smalltalk. Its syntax looks deceptively simple at first sight. In fact, Ruby is a good deal more complex than it may appear. The Ruby language was created by Yukihiro Matsumoto (commonly known as 'Matz') and it was first released in 1995.

What Is Rails?

Rails is a web development framework (a collection of code libraries and software tools) that uses Ruby as its programming language. It is popularly known as 'Ruby On Rails'. While Rails is an impressive framework, it is not the be-all and end-all of Ruby. Indeed, if you decide to leap right into Rails development without first mastering Ruby, you may find that you end up with an application that you don't even understand. *The Little Book of Ruby* is not about Rails; it concentrates entirely on the Ruby programming language. However, if you intend to program Rails applications, this book will give you the grounding you need in order to understand Rails code and write your own custom Ruby On Rails applications.

Installing Ruby

Before doing anything else, you need to install Ruby on your computer. If you are using Mac OS X (or some versions of Linux) you may already have Ruby installed. At any rate, you can download the latest version of Ruby from www.ruby-lang.org. Be sure to download the binaries (not merely the source code). For Windows users, the easiest way to set up Ruby on your system is by using the Ruby Installer for Windows available from: http://rubyinstaller.org/

Ruby 1 or Ruby 2?

At the time of writing, the latest version of Ruby is Ruby 2.x where 'x' represents the minor version numbers used differentiate specific updates to Ruby 2 (e.g. Ruby 2.1.6, Ruby 2.2.2 and so on). Many programmers are still using earlier versions of Ruby, however, such as Ruby 1.8 or 1.9. In most cases, the syntax and behaviour of Ruby 1.x (from at least version 1.8) and Ruby 2.x are identical. That is not to say that there are no important differences but at this stage they need not concern us. In most cases the code described in this book will run identically in Ruby 1.8, 1.9 and 2.x.

Installing A Ruby Editor or IDE

While you could write Ruby programs in a simple text editor and run them from the command prompt, it is a good idea to use (at the very minimum) a Ruby editor with syntax code coloring. If you are really serious about Ruby you should think about using a Ruby IDE that offers features such as IntelliSense (code completion) and integrated debugging. Here are a few possibilities…

SciTE

Free programmers' editor with some basic support for Ruby editing:
http://www.scintilla.org/SciTE.html

Komodo Edit

Free editor for multiple programming languages including Ruby. A more powerful commercial edition, Komodo IDE, is also available. Runs on Windows, Linux, Mac OS X:
http://komodoide.com/komodo-edit/

RadRails

A free Eclipse-based IDE primarily for Rails-based Ruby development:
http://www.aptana.com/products/radrails/

RubyMine

Commercial cross-platform IDE that is tailored for Ruby On Rails users. Runs on Windows, Linux, Mac OS X:

http://www.jetbrains.com/ruby/

TextMate

Commercial programmers' editor, with Ruby coding features, for Mac users:

http://macromates.com/

Sapphire

Commercial IDE (the successor to *Ruby In Steel*) for Microsoft Visual Studio:

http://www.sapphiresteel.com

Get The Source Code Of The Sample Programs

All the programs in every chapter in this book are available as a Zip archive which may be downloaded from:
http://www.sapphiresteel.com/ruby-programming/The-Little-Book-Of-Ruby.html.

When you unzip the programs you will find that they are grouped into a set of directories – one for each chapter. If you are using *Sapphire* or *Ruby In Steel*, you will be able to load all the programs as a single solution, 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 you may load the individual Ruby programs one at a time.

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 helloworld.rb
```

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

Running Ruby From a Command Prompt

On Windows

Click the *Start* menu. In the *Run* or *Search* text entry field enter:

cmd

This should open a text-mode window. You can now change to the directory containing the Ruby program you wish to run. To change directory enter **cd** followed by the directory name. e.g. **cd C:\myrubyprograms**

On Mac OS X

You need to open a 'Terminal' window. Double-click your hard-drive (e.g. *Macintosh HD*). Open the *Applications* folder. Then open the *Utilities* folder. Double-click the *Terminal* icon. This will open a window into which you can enter commands to run Ruby. To change directory enter **cd** followed by the directory name. e.g. **cd /myrubyprograms.**

TIP: OPEN A COMMAND WINDOW IN THE SELECTED FOLDER

On Windows and OS X it is possible to navigate to a folder using the system File Browser and open a Command Window or Terminal on that folder. This is much simpler and quicker than navigating directories from the system prompt. This is how to do that:

Windows

Open the Windows Explorer. Navigate to a specific directory. Enter `CMD` in the Explorer address bar. This opens a command window in the selected directory.

OS X

Before you can do this on a Mac you need to set an option as follows: Go to *System Preferences*, then select *Keyboard* and *Services*. In the Keyboard settings dialog, scroll down to the entry labelled 'New Terminal at Folder'. Check it. Now in Finder you can navigate to the folder you need (the one containing your Ruby program) – Right click the folder. The popup menu will contain an item called '*New Terminal at Folder*', When selected this opens a Terminal on the current folder.

How To Use This Book

This book is a step-by-step tutorial to programming in Ruby and you can follow it chapter by chapter, reading the text and running the sample programs. On the other hand, if you prefer to 'dip in', you may want to try out some of the programs in whichever order takes your fancy; then refer back to the text for explanations. There are no monolithic applications in this book – just small, self-contained sample programs – so it's easy to skip from chapter to chapter if you wish…

Making Sense Of The Text

In **The Little 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 little box 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 shaded 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…!

Chapter One

IN WHICH WE TIE UP SOME STRINGS, ADD UP SOME NUMBERS, MESS ABOUT WITH METHODS AND DIAGNOSE SOME CONDITIONS…

From the fact that you are reading this, it is safe to deduce that you want to program Ruby – and, if you are anything like me, you will be impatient to get on with it. OK, let's not hang around. I'll assume that you already have Ruby installed. If not, you'll need to do that first, as explained in the Introduction…

Now, let's start coding. Fire up your editor and enter the following:

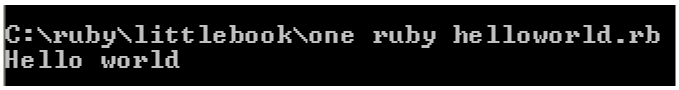
helloworld.rb

```
puts 'Hello world'
```

Here `puts` is the name of a function or 'method' that can display or 'put' a string such as 'Hello world'. Run this program (as explained earlier). If you are using an editor which lacks an interactive console, you should run this program from the command prompt. To do this, open a command window or Terminal and navigate to the directory containing the source code then enter **ruby** followed by the program name, like this:

ruby helloworld.rb

All being well, Ruby should display: `Hello world`.



```
C:\ruby\littlebook\one ruby helloworld.rb
Hello world
```

This must just about the shortest 'Hello world' program in the history of programming so we'll immediately move on to a way of getting input from the user…

The obvious next step is to 'get' a string. The Ruby method for this is `gets`.

uppercase.rb

OBJECTS AND METHODS

Before we go any further, let me explain what a *method* is. To understand that, we also need to be clear on what an *object* is.

Ruby is a highly Object Oriented programming (OOP) language. Everything from an integer to a string is considered to be an object. And each object has built in functions or 'methods' which can be used to do various useful things. To use a method, you generally need to put a dot after the object name, then append the method name. For example, Here I am using the `upcase` method to display the string, "hello world" in uppercase:

```
puts( "hello world".upcase )
```

But some methods such as `puts` and `gets` are available everywhere and don't need to be associated with a specific object. Technically speaking, these methods are provided by Ruby's `Kernel` module and they are *included* in all Ruby objects (Modules and inclusion are explained in Chapter Nine). When you run a Ruby application, an object called `main` is automatically created and this object provides access to the `Kernel` methods.

The **helloname.rb** program prompts the user for his or her name – let's suppose it's "Fred" - and then displays a greeting: "Hello Fred". Here's the code:

helloname.rb

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

While this is still very simple, there are a few 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 predeclared this variable, nor have I specified its type. In Ruby you can create variables as you need them and Ruby 'infers' their types. Here 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.

Incidentally, the parentheses following `gets()` are optional as are the parentheses (or 'round brackets') enclosing the strings after `print` and `puts`; the code would run just the same if you removed the parentheses. Round brackets help to avoid potential ambiguity in code and, in some cases, the Ruby interpreter will warn you if you omit them.

While some Ruby programmers like to omit parentheses whenever possible, I am not one of them; you will, therefore, find parentheses used liberally in my programs.

Strings and Embedded Evaluation

The last line in the **helloname.rb** program 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.

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 double-quoted string shown below would, in the process of evaluation, call the `showname` method and, as a result, it would display the string "Hello Fred":

string_eval.rb

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

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

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

Run the **string_eval.rb** program to see if you were right.

COMMENTS

Lines beginning with a # character are treated as comments (they are ignored by the Ruby interpreter):

```
# This is a comment
```

Methods

A *method* is so called because it provides a method (that is, 'a way') for an object to respond to messages. In OOP terminology, you send a message to an object by asking it to do something. So let's imagine that you have an object called ob which has a method called saysomething. This is how you would send a saysomething message to the object:

object.rb

```
ob.saysomething
```

Let's suppose that the saysomething method looks like this:

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

The result is, that when you send ob a saysomething message it responds by running the saysomething method which displays "Hello".

OK, so that's the 'pure OOP' way of describing this. A not-so-pure OOP way of describing it would be to say that `saysomething` is like a function or subroutine which is bound to the object and can be called using dot notation: `ob.saysomething`.

method.rb

In Ruby a method is declared with the keyword `def` followed by a method name which should begin with a lowercase letter, like this:

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

You may optionally put one or more arguments, separated by commas, after the method name:

```
def showname( aName )
   puts( "Hello #{aName}" )
end

def return_name( aFirstName, aSecondName )
   return "Hello #{aFirstName} #{aSecondName}"
end
```

The parentheses around the arguments are optional. The following syntax is also permissible:

```
def return_name2 aFirstName, aSecondName
   return "Hello #{aFirstName} #{aSecondName}"
end
```

As explained previously, for the sake of clarity, I am very much prejudiced in favour of parentheses but you can omit them if you wish.

mainob.rb

If methods belong to objects, which object owns any 'free-standing' methods (such as `showname` in **method.rb**) that you write in your code? As I mentioned earlier, Ruby automatically creates an object named `main` when you run a program and it is to this object that any free-standing methods belong.

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 code would do the calculation and display the result:

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

Obviously, it 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 Tax 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 `to_f` is a method of the `String` class (and the variable `s` is assigned a string entered by the user). The `to_f` method 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.

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:

tax_calculator.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 parentheses 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
```

However, the `end` keyword that terminates the `if` block is *not* optional. Forget to add it and your code will not run.

Chapter Two

DEFINING CLASSES, CREATING OBJECTS AND PEEKING INSIDE THEM…

So far we've used a number of 'standard' Ruby objects such as numbers and strings. Let's now see how to create new types of objects of our very 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. This is a very simple class:

```
class MyClass
end
```

And this is how I would create a usable object from it:

```
ob = MyClass.new
```

Not that I can do a great deal with my `ob` object – for the simple reason that I haven't written any Ruby code in the `MyClass` class, from which it is created.

object_class.rb

Actually, if you create an 'empty' class like `MyClass`, the objects created from it will not be totally useless. All Ruby classes automatically inherit the features of the `Object` class. So my `ob` object can make use of `Object` methods such as `class` (which tells an object display its class).

Try this:

```
puts ob.class
```

When run, this code displays the following:

```
MyClass
```

To make MyClass a bit more useful, I need to give it a method or two. In this example (which was mentioned briefly in the last chapter), I've added a method called saysomething:

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

Now, when I create a MyClass object, I can call this method in order to get that object to say "Hello":

```
ob = MyClass.new
ob.saysomething
```

And this is the output:

```
Hello
```

Instances and Instance Variables

Let's create some more useful objects. No home (or computer program) should be without a dog. So let's make ourselves a Dog class:

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

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

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 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 (remember that 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 names to the dogs, I need to have some way to find out their names later on. Each dog needs to know its own name, so let's give it a `get_name` method:

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

The `return` keyword here is optional. Ruby methods will always 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:

dogs_and_cats.rb

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

  def get_name
     return @myname
  end

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

Now, I 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)
```

For the sake of variety – and to show that I am not biased against our feline friends – I have added a `Cat` class in my program, **dogs_and_cats.rb**. The `Cat` class is similar to the `Dog` class apart from the fact that its `talk` method, naturally enough, returns a *miaow* instead of a *woof*.

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…

Constructors – new and initialize

treasure.rb

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

The `Treasure` class has a few interesting extras, however. First of all, it 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:

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

When a class contains a method named `initialize` this is 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, 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).

NOTE: 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.

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 class 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.

OBJECT AND BASICOBJECT

I said earlier that the `Object` class is the ultimate ancestor of all other classes in Ruby. In versions of Ruby prior to Ruby 1.9 that was literally true. However, in Ruby 1.9 and later, the `Object` class is the descendent of another class called `BasicObject`. The Ruby class library documentation describes `BasicObject` as a "blank class" which can be used to create "object hierarchies independent of Ruby's object hierarchy". As this is something we definitely do not want to do, for now it is safe to consider `Object` as the base class or "root" of all the other classes we shall ever use.

Inspecting Objects

Incidentally, in the **treasure.rb** program you will find 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 this program I have called `t1.inspect` by evaluating it inside a double-quoted string as I explained in the last chapter:

```
puts "Inspecting 1st treasure: #{t1.inspect}"
```

Running the code shown above displays something like this:

```
Inspecting 1st treasure: #<Treasure:0x2a94818 @name="Sword",
@description="an Elvish weapon forged of gold">
```

This begins with the class name, `Treasure`; this is followed by a number such as `0x2a94818` – this is Ruby's internal identification code for this particular object; then there are the names and values of the object's variables.

p.rb

Ruby provides the `p` method as a shortcut to inspect and display objects:

```
p( anobject )
```

The actual output will depend on the type of object being inspected. For example, if the object is an integer, the integer itself is displayed. If it is a string, the string (including a pair of string delimiters – the quotation marks) will be displayed:

```
a = "hello"
b = 123
p( a )
p( b )
```

This code produces the following output:

```
"hello"

123
```

to_s.rb

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 **to_s.rb** program.

As you will see when you run this program, *classes* such as `Class`, `Object`, `String` and `Treasure` simply return their names when the `to_s` method is called:

```
puts(Treasure.to_s)
```

This displays:

```
Treasure
```

But an *object*, such as the `Treasure` object, `t`, returns its identifier:

```
puts(t.to_s)
```

This displays:

```
t.to_s: #<Treasure:0x2985120>
```

The identifier displayed by `to_s` is the *same identifier* shown by the `inspect` method.

```
puts(t.inspect)
```

This displays:

```
t.inspect: #<Treasure:0x2985120 @name="Sword", @description="A
lovely Elvish weapon">
```

NOTE: When you run this code the *actual* number, such as `0x2985120` may be different from the one shown above. That is because the number is Ruby's internal identifier for the object and this will be created anew whenever the code is run.

Looking over my **treasure.rb** program I can't help thinking that its code is a bit repetitive. After all, why have a `Thing` class which contains a name and a `Treasure` class which also contains a name (that is, the `@name` instance variable), each of which are coded independently? It would make more sense to regard a `Treasure` as a 'type of' `Thing`. If I were to develop this program into a complete adventure 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 lesson…

Chapter Three

CLASS HIERARCHIES…

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

Creating a Class Hierarchy

It makes sense to create a class hierarchy in which a class which is a 'special type' of some other class simply 'inherits' the features of that other class. 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.

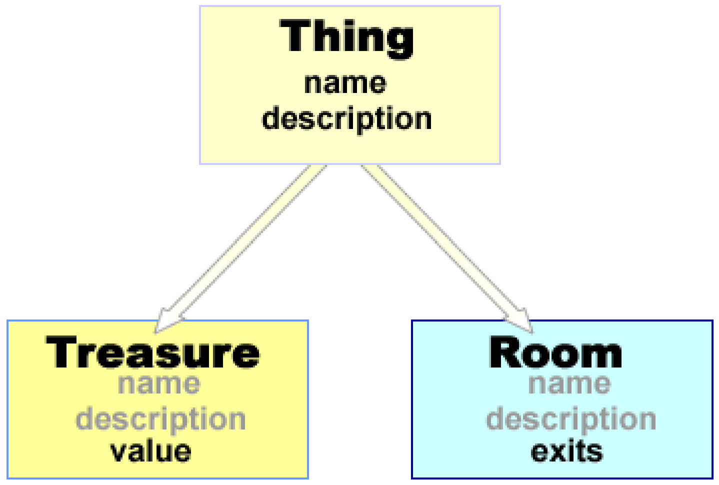
Class Hierarchies – Ancestors and Descendants

In this book, I often talk about 'descendant' classes 'inheriting' features from their 'ancestor' classes. These terms deliberately suggest a kind a family tree of 'related' classes. In Ruby, each class only has one parent. A class may, however, descend from a long and distinguished family tree with generations of 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. The `Treasure` class 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...



The diagram above 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*.

adventure1.rb

Let's see how to create a descendant class in Ruby. Load up the **adventure1.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.

"ENCAPSULATION" is a term that refers to the 'modularity' of an object. Put simply, it means that only the object itself can mess around with its own internal state. The outside world cannot. The benefit of this is that the programmer is able to change the implementation of methods without having to worry that some external code elsewhere in the program relies upon some specific detail of the previous implementation.

In order to obtain the value of each variable in a `Thing` object 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`:

```
def get_name
    return @name
end

def set_name( aName )
    @name = aName
end
```

Superclasses and Subclasses

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

```
class Treasure < Thing
```

Here 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 `get_name`, `set_name`, `get_description` and `set_description` methods 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 (with `new`), its `initialize` method is automatically called. A `Treasure` object has three variables to initialize (`@name`, `@description` and `@value`), so its `initialize` method takes three arguments:

```
def initialize( aName, aDescription, aValue )
```

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* in the ancestor or 'super' class.

The current method in the `Treasure` class is called `initialize` so when code inside this method passes and the two arguments `aName` and `aDescription` to `super` it is actually passing them to the `initialize` method of its superclass, `Thing`.

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.

Chapter Four

ACCESSORS, ATTRIBUTES AND CLASS VARIABLES…

Now, getting back to the little adventure game work I was programming earlier on… I still don't like the fact that the classes are full of repetitive code due to all those *get* and *set* accessors. Let me see what I can do to remedy that.

Accessor Methods

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, I need to modify the `Treasure` class definition. One way of doing this would be by rewriting the accessor methods for `@description` as follows:

```
def description
   return @description
end

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

accessors.rb

I have added accessors similar to the above in the **accessors.rb** program. There are two differences from my previous version. First, both of the accessors are called `description` rather than `get_description` and `set_description`; secondly the *set* accessor appends an equals sign (`=`) to the method name. 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 )
```

NOTE: When you write a *set* accessor in this way, you must append the `=` character directly 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 but outside of any methods, like this:

```
class Thing
  attr_reader :description
  attr_writer :description

  # some methods here…
end
```

SYMBOLS

In Ruby, a symbol is a name preceded by a colon. `Symbol` is defined in the Ruby class library to represent names inside the Ruby interpreter. Symbols have a number of special uses. For example, when you pass one or more symbols as arguments to `attr_reader` (while it may not be obvious, `attr_reader` is, in fact, a method of the `Module` class), Ruby creates an instance variable and a *get* accessor method to return the value of that variable; both the instance variable and the accessor method will have the same name as the specified symbol.

Calling `attr_reader` with a symbol has the effect of creating an instance variable with a name matching the symbol and a *get* accessor for that variable. Calling `attr_writer` similarly creates an instance variable with a *set* accessor. Here, the variable would be called `@description`. Instance variables are considered to the 'attributes' of an object, which is why the `attr_reader` and `attr_writer` methods are so named.

accessors2.rb

The **accessors2.rb** program contains some working examples of attribute readers and writers in action. Notice that the `Thing` class defines a short-form *set* accessor (using `attr_writer` plus a symbol) for the `@name` variable:

```
attr_writer :name
```

But it has a long-form *get* accessor – an entire hand-coded method – for the same variable:

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

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` class's `capitalize` method to return the string value of `@name` with its initials letters in uppercase.

The `@description` attribute needs no special processing at all so I have used both `attr_reader` and `attr_writer` to get and set the value of the `@description` variable.

ATTRIBUTES OR PROPERTIES?

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

When you want 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
```

Attributes Create Variables

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:

```
class Treasure < Thing
      attr_accessor :value

      def initialize( aName, aDescription )
          super( aName, aDescription )
      end
end
```

The only indication that an `@value` variable exists at all is this accessor definition which declares a `value` attribute:

```
attr_accessor :value
```

My code down at the bottom of the source file sets the value of each Treasure object:

```
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}"
```

And this is the output, showing the `@value` variable inside the objects:

```
This is treasure1: #<Treasure:0x297c5e8 @name="sword",
@description="an Elvish weapon forged of gold (now somewhat
tarnished)", @value=100>


This is treasure2: #<Treasure:0x297c5a0 @name="dragon horde",
@description="a huge pile of jewels", @value=500>
```

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, parentheses around the arguments are optional.

adventure2.rb

Now let's see how to put attribute readers and writers to use in my adventure game. Load up the **adventure2.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 to_s method in the Thing class 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 to call the superclass's method, 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.

In some Ruby books, a hash or pound sign may be shown between the class name and a method name like this:

`Treasure#initialize`.

This is purely a convention of documentation (one which I prefer to ignore) and is not real Ruby syntax.

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
```

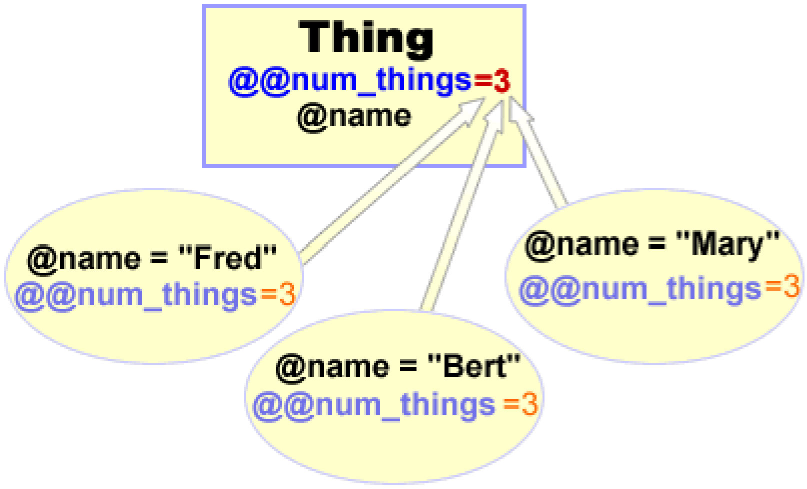
The two `@` characters at the start of this variable name, `@@num_things`, define this to be a 'class variable'. The variables we've used inside classes up to now have been *instance* variables, preceded by a single `@`, like `@name`. Whereas each new object (or 'instance') of a class assigns its own 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 (it uses += to add 1 to it) in the 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 (a sequential list) 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 shortly.

Objects 'share' Class 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.

Find the code down at the bottom of the file **adventure2.rb** 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.

Chapter Five

ARRAYS…

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.

Introducing Arrays

array0.rb

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 hold 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 **adventure2.rb** in chapter 4 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])
puts(arr[1])
puts(arr[2])

puts(arr[3])
```

This produces the following output:

```
"a"
"b"
"c"
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. 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 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:

```
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')
```

Or you could 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.

array_new.rb

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 parentheses, Ruby considers it a syntax error if you fail to leave a space between the `new` method and the opening square bracket.

These are all correct:

```
arrayob = Array.new([1,2,3])
arrayob = Array.new( [1,2,3] )
arrayob = Array.new [1,2,3]
```

But this is an error:

```
arrayob = Array.new[1,2,3]
```

This is another good reason for making a firm habit of using parentheses when passing arguments!

multi_array.rb

For some 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]
```

ITERATORS AND FOR LOOPS

The code inside a `for` loop is executed for each element in an expression. The syntax is 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
```

Indexing Into Arrays

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 (values between a start index and an end index separated by two dots):

array_index.rb

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

print( arr[0,5] )
print( arr[-5,5 ] )
print( arr[0..4] )
print( arr[-5..-1] )
```

This code produces the following output:

```
["h", "e", "l", "l", "o"]
["w", "o", "r", "l", "d"]
["h", "e", "l", "l", "o"]
["w", "o", "r", "l", "d"]
```

Notice that, as with strings, when provided with 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]
```

This code produces the following output:

```
["h", "e", "l", "l", "o"]
```

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:

array_assign.rb

```
arr = []

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

If I now inspect and print `arr`:

```
p( arr )
```

This is what I see:

```
[[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'
```

If I now inspect and print `arr2`:

```
p( arr2 )
```

This is what I see:

```
["H", "e", "L", "L", "O", "-", "W", "a", "l", "d", "o"]
```

Chapter Six

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) and a value (in a dictionary, this would be the definition of the word).

Creating Hashes

hash1.rb

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. 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'
}
```

UNIQUE KEYS?

Take care when assigning keys to Hashes. If you use the same key twice in a `Hash`, you will end up over-writing 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",
"treasure3"=>"Ruby ring"}
```

Indexing Into A Hash

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

```
puts(h1['room2'])
```

This displays:

```
"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'])
p(h2['unknown_treasure'])
```

This displays:

```
nil

"Some kind of ring"
```

Use the `default` method to get the default value and the `default=` method to set it (see Chapter 4 for more information on *get* and *set* methods):

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

Hash Operations

hash2.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: (note, the data shown on the shaded lines after the code samples show the values returned when each piece of code is run) :

```
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"]
```

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

```
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. 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]
```

Chapter Seven

LOOPS AND ITERATORS…

Much of programming is concerned with repetition. You may want a program to beep 10 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 have to 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 Chapter Five (arrays) 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. Indeed, the author of Ruby describes for as "syntax sugar" for the each method which is implemented by Ruby's 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 the for loop 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
```

Which can be rewritten using each:

```
(1..10).each do |i|
 puts(i)
end
```

A range expression such as 1..3 must be enclosed between parentheses 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 parentheses are optional when a range is used in a for loop.

When iterating over items using each the block of code between do and end is called (predictably, perhaps?) an 'iterator block'.

Block Parameters

In Ruby 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 (for example, an array).

Blocks

Ruby has an alternative syntax for delimiting blocks. You may use `do` and `end`, like this...

block_syntax.rb

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

Or you can use curly braces `{ }` like this:

```
# 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.

A **Boolean** condition is one that evaluates to either *true* or *false*.

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
```

This is an example showing the various alternative syntaxes:

while.rb

```
$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.

while2.rb

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.

To appreciate the differences in behaviour of these two types of `while` loop, run **while2.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 they 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 there is the option of enclosing 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
```

Chapter Eight

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. The little test above ensures that `subtotal` is never a minus figure. If the user, in a fit of madness, enters a value less than 0, the `if` test spots this since the condition `(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..else

if_else.rb

A simple `if` test 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 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:

```
working_overtime = true

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

and_or.rb

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 parentheses around any code to be evaluated as a single unit, like this:

```
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 previous example, I 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.

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.

if_elsif.rb

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 the `chomp()` method 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:

```
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
```

This code has a bug. It asks for a number between 1 and 1000 but it accepts other numbers. See if you can rewrite the tests to fix this!

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 `and`s and `or`s) you should enclose it in parentheses.

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'
```

unless

unless.rb

Ruby also can also perform `unless` tests, which are the 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 in Chapter 7. 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, for example, you repeatedly need to take some well-defined action if some condition is true.

This is how you might pepper your code with debugging output if a constant called `DEBUG` is true:

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

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

The `then` keyword can be omitted if the test and the code to be executed are on separate lines, like this:

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

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 then puts("It's Friday" )
            puts("...nearly the weekend!")
      when 6 then puts("It's Saturday!" )
            # the following never executes
      when 5 then 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 then puts( "Yippee! It's the weekend! " )
```

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' then puts( "Yup, '#{i}' is Monday" )
```

case2.rb

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

```
case( i )
    when 1 then puts("It's Monday" )
    when 2 then puts("It's Tuesday" )
    when 3 then puts("It's Wednesday" )
    when 4 then 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 then puts( "It's Friday all over again!" )
          else puts( "That's not a real day!" )
    end
```

Chapter Nine

MODULES AND MIXINS…

As mentioned in an earlier chapter, each Ruby class can only have 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, also have many other parents and children, you risk ending up with an impenetrable network rather than the neat, well-ordered hierarchy which you may have intended.

Nevertheless, there are occasions when it is useful for a class to be able to implement features which it has in common with more than one other pre-existing class.

For example, a Sword might be a type of Weapon but also a type of Treasure; a House might be a type of Building but also a type of Investment and so on.

A Module Is Like A Class…

Ruby's solution to this problem is provided by Modules. At first sight, a module looks very similar to a class. Just like a class it 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`. To turn this into a class you would only need to replace the word `module` in its definition with the word `class`.

Module Methods

In addition to instance methods a module may also have module methods which are preceded by 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.

Which leads us to the next question: if you can't create an object from a module, what are modules for?

This question can be answered in two words: **namespaces** and **mixins**. Ruby's 'mixins' provide a way of dealing with the little problem of multiple inheritance which I mentioned earlier. 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' - which means that 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`.

Constants

Constants are like variables except their values do not (or *should* not!) change. In fact, it is (bizarrely!) possible to change the value of a constant in Ruby but this is certainly not encouraged and Ruby will warn you if you do so. Note that constants begin with a capital letter.

modules1.rb

Let’s assume we have this module:

```
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 constants using `::` like this:

```
puts(MyModule::GOODMOOD)
```

We can similarly access module methods using dot notation – that is, specifying the module name followed by a full stop and the method name. The following would print out “I'm grumpy. How are you?”:

```
puts( MyModule.greet )
```

Module 'Instance Methods'

This just leaves us with the problem of how to access the instance method, `greet`. As the module defines a closed namespace, 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 would, 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, as I said earlier, you cannot create instances of modules. So how the heck can we use their instance methods? This is where those mysterious mixins enter the picture…

Included Modules or 'Mixins'

modules2.rb

An object can access the instance methods of a module just 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:

```
include MyModule
puts( greet )
```

The process of including a module in a class is also called 'mixing in' the module – which explains why included modules are often called 'mixins'.

When you include objects into a class definition, any objects created from that class will be able to use the instance methods of the included module just as though they were defined in the class itself.

modules3.rb

```
class MyClass
  include MyModule

  def sayHi
    puts( greet )
  end

  def sayHiAgain
    puts( MyModule.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, like this:

```
ob = MyClass.new
ob.sayHi
ob.sayHiAgain
puts(ob.greet)
```

In short, then, modules can be used as a means of grouping together related methods, constants and classes within a named scope. In this respect, 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 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 methods such as `deadliness` and `power`), but it also needs to have the methods of a Treasure (such as `value` and `insurance_cost`). If you define these methods inside a `Treasure` ***module*** rather than a `Treasure` ***class***, the `Sword` class would be able to *include* the `Treasure` module in order to add ('mix in') the `Treasure` methods to the `Sword` class's own methods.

In fact, it could mix in more than one module if it needed to access methods from those too. Here, for example, my `Sword` class descends from the `Weapon` class and mixes in the `Treasure` and `MagicThing` modules (it then adds a `name` attribute):

modules4.rb

```
class Sword < Weapon
  include Treasure
  include MagicThing

  attr_accessor :name
end
```

Note, incidentally, that any variables which are local to a 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. The **mod_vars.rb** program illustrates this.

mod_vars.rb

```
x = 1            # x is local to this program

module Foo
    x = 50       # x is local to module Foo

    def no_bar
      return x
    end

    def bar
      @x = 1000
      return  @x
    end

    puts( "In Foo: x = #{x}" )   # this is the module-local x (50)
end

include Foo

puts(x)          # this is the ‘program-local’ x (1)
puts(bar)
puts( no_bar )   # ERROR: Cannot access module-local variable x needed by
                 # the no_bar method
```

This code produces the following valid output:

```
In Foo: x = 50

1

1000
```

But when an attempt is made to execute the `no_bar` method an error message is displayed:

```
`no_bar': undefined local variable or method `x' for main:Object
(NameError)
```

Including Modules From Files

So far, we've mixed in modules which have all been defined within a single source file. Often it is more useful to define modules in separate files and include them 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:

requiremodule.rb

```
require( "./testmod.rb" )
```

The required file must be in the specified directory (in the example above `"./"` indicates the current directory or 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"
```

So, if I wanted to add the current directory to the paths searched by Ruby and then require files from that directory using the file names only, I could rewrite my code like this:

```
$: << "./"
require( "testmod.rb")
```

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" ))
```

REQUIRE_RELATIVE

In older versions of Ruby (prior to 1.9), you could require files from the current directory without having to include the `"./"` path. In Ruby 1.9 and Ruby 2 you can do this using the alternative method `require_relative` like this:

```
   require_relative( "testmod.rb")
```

Pre-Defined Modules

The following modules are built into the Ruby interpreter:

```
   Comparable, Enumerable, FileTest, GC, Kernel, Math, ObjectSpace,
   Precision, Process, Signal
```

The most important of the pre-defined modules is `Kernel` which, as mentioned earlier, 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.

Chapter Ten

GOING FURTHER…

The time has now come to wrap up this *Little Book Of Ruby*. Let's do that by looking at one more sample project – a little CD database which lets you create new objects (one for each disc in your CD collection), add them to an array and store them on disk.

Saving Data

In order to save the data to disk I have used Ruby's YAML library:

data_save.rb

```
# saves data to disk in YAML format
def saveDB
    File.open( $fn, 'w' ) {
        |f|
        f.write($cd_arr.to_yaml)
    }
end
```

YAML

YAML describes a format for saving data as human-readable text. The data can be subsequently reloaded from disk in order to reconstruct the array of CD objects in memory:

```
def loadDB
    input_data = File.read( $fn )
    $cd_arr = YAML::load( input_data )
end
```

Much of the coding in this little program should be familiar from our previous projects. A couple of things need to be highlighted, however.

First, variables beginning with a dollar `$` are 'global' so are usable by all the code throughout the program (recall that instance variables, starting with `@`, are only usable within the confines of a specific object; while local variables, starting with a lowercase letter, are only usable within a well-defined 'scope' such as within a specific method).

Files

Also notice that we use the `File` class to check if a file exists:

```
if File.exist?( $fn )
```

Here, `exist?` is a 'class method' – that is, it 'belongs to' the `File` class rather than to an instance of the `File` class. That explains how we can invoke the method from `File` itself rather than having to invoke it from a new `File` object. This may remind you of the module methods discussed in Chapter Nine – another example of the similarities between modules and classes.

Moving On...

In conclusion, I hope you've enjoyed this little introduction to the Ruby language and that it may be just the start of many years of enjoyable and productive Ruby development.

If you want to take your study of Ruby further, you may want to get a copy of my more advanced Ruby programming guide, **The Book Of Ruby**, available as a paperback from No Starch Press:

http://www.nostarch.com/boruby.htm

Alternatively (*if you haven't already done so!*) you can subscribe to one of the **Ruby courses** which I teach online:

https://www.udemy.com/u/huwcollingbourne/

And, to keep up to date with any new courses and updates, be sure to visit the **Bitwise Courses web site**:

http://www.bitwisecourses.com/

Our **Facebook** page:

https://www.facebook.com/BitwiseCourses

Our **Twitter** Feed:

https://twitter.com/bitwisecourses

And our **YouTube channel**:

https://www.youtube.com/user/BitwiseCourses

Good programming!