Ruby
Essentials

Ruby Essentials

Ruby Essentials – Version 1.12

Find more technology eBooks at www.ebookfrenzy.com

Table of Contents

Chapter 1.   About Ruby Essentials

Ruby is a flexible and intuitive object-oriented programming language. From modest beginnings in Japan where it rapidly gained a loyal following, the popularity of Ruby has now spread throughout the programming world.

This surge in popularity can, in no small part, be attributed to the introduction and wide adoption of the Ruby on Rails framework. It is difficult, however, to get the most out of Ruby on Rails without first learning something about programming in Ruby, and this is where Ruby Essentials comes in.

Ruby Essentials is intended to provide a concise and easy to follow guide to learning Ruby. Everything from installing Ruby and the basics of the language through to topics such as arrays, file handling and object-oriented programming are covered, all combined with easy to understand code examples which serve to bridge the gap between theory and practice.

Ruby Essentials is designed to be of equal use both to those experienced in other programming languages and to novices who have chosen Ruby as their "first programming language".

Chapter 2.   What is Ruby?

In this chapter of Ruby Essentials we will learn about what Ruby is, how it came into existence and what it is useful for.

2.1   The History of Ruby

Ruby was created by Yukihiro Matsumoto (more affectionately known as *Matz*) in Japan starting in 1993. Matz essentially kept Ruby to himself until 1995 when he released it to the public. Ruby quickly gained a following in Matz's home country of Japan in the following years, and finally gained recognition in the rest of the programming world beginning in the year 2000. From that point on Ruby has grown in popularity, particularly because of the popularity of the Ruby on Rails web application development framework.

2.2   What is Ruby?

Ruby is an object-oriented interpreted scripting language. When we say it is interpreted we mean to say that the Ruby source code is compiled by an interpreter at the point of execution (similar in this regard to JavaScript and PHP). This contrasts with compiled languages such as C or C++ where the code is pre-compiled into a binary format targeted to run on a specific brand of microprocessor.

There are advantages and disadvantages to being an interpreted language. A disadvantage is speed. Because the source code has to be interpreted at runtime this means that it runs slower than an equivalent compiled application. A secondary concern for some is the fact that anyone who uses your application will also be able to see the source code. In the world of open source this is less of a problem than it used to be, but for some proprietary applications this might prove to be unacceptable.

The primary advantage of interpreted languages is that they are portable across multiple operating system platforms and hardware architectures. A compiled application, on the other hand, will only run on the operating system and hardware for which it was compiled. For example, you can take a Ruby application and run it without modification on an Intel system running Linux, an Intel system running Windows, an Intel system running Mac OS X or even a PowerPC system running Mac OS or Linux. To do this with a C or C++ application you would need to compile the code on each of the 5 different systems and make each binary image available. With Ruby you just supply the source code.

Another advantage of being interpreted is that we can write and execute Ruby code in real-time directly in the Ruby interpreter. For those who like to try things out in real time (and not everyone does), this is an invaluable feature.

2.3   Why is Ruby so Popular?

Firstly, Ruby is a very intuitive and clean programming language. This makes learning Ruby a less challenging task than learning some other languages. Ruby is also a great general purpose language. It can be used to write scripts in the same way you might use Perl and it can be used to create full scale, standalone GUI based applications. Ruby's usefulness doesn't end there however. Ruby is also great for serving web pages, generating dynamic web page content and excels at database access tasks.

Not only is Ruby intuitive and flexible it is also extensible, enabling new functionality to be added through the integration third-party, or even home grown libraries.

And, of course, being an interpreted language means that Ruby is portable. Once an application has been developed in Ruby it will run equally well on Ruby supported platforms such as Linux, UNIX, Windows and Mac OS X.

Chapter 3.   Getting and Installing Ruby

No matter how wonderful Ruby is, there isn't much you can do with it if it is not installed on your computer system. In this chapter we will cover the download and installation of Ruby on Linux, UNIX and Windows.

Ruby is itself written in the C programming language. This means that either a binary distribution for your chosen operating system and hardware platform needs to be installed, or the Ruby sources need to be downloaded and compiled on your target system. Whilst compiling Ruby yourself might be fun, it usually makes more sense to simply download and install one of the many pre-built Ruby packages rather than attempt to build your own. In this chapter we will cover installing pre-built Ruby packages on each platform.

3.1   Installing Ruby on Linux

There are a number of different Linux distributions available today and it makes sense to install the Ruby package built specifically for your chosen Linux flavor. The best way to do this is to use the standard package manager for that particalur Linux.

3.1.1   Ruby on Red Hat Enterprise and Fedora Linux

Red Hat Enterprise Linux and Fedora Linux both use the *YUM* installation manager and the Red Hat Package Manager (*rpm*). The first step is to verify if Ruby is already installed. This can be achieved using the following rpm command. In this example, Ruby is not yet installed:

```
rpm -q ruby
package ruby is not installed
```

If Ruby is not installed, it can be installed using the *yum* update manager. This needs to be performed as *root* so the *super-user* password will be required in the following steps:

```
su -
yum install ruby
```

The *yum* tool will locate the *ruby* package and any other packages on which Ruby is dependent and prompt you to install the packages:

```
Downloading Packages:
(1/2): ruby-1.8.1-7.EL4.8 100% |                              | 156 kB    00:10
```

```
(2/2): ruby-libs-1.8.1-7. 100% |                              | 1.5 MB    01:23
Running Transaction Test
Finished Transaction Test
Transaction Test Succeeded
Running Transaction
  Installing: ruby-libs                     ######################### [1/2]
  Installing: ruby                          ######################### [2/2]


Installed: ruby.i386 0:1.8.1-7.EL4.8
Dependency Installed: ruby-libs.i386 0:1.8.1-7.EL4.8
Complete!
```

Once the installation is complete, you may re-run the *rpm* command to verify the package is now installed:

```
rpm -q ruby
ruby-1.8.1-7.EL4.8
```

Alternatively, you can verify that Ruby is installed by running it with the command line option to display the version information:

```
ruby -v
ruby 1.8.1 (2003-12-25) [i386-linux-gnu]
```

3.1.2 Ruby on Ubuntu and Debian Linux

Debian, Ubuntu and other Debian derived Linux distributions use the *apt-get* tool to manage package installation. If you are running Ubuntu Linux and get the following output from the *ruby* command, you need to install Ruby:

```
$ ruby
The program 'ruby' is currently not installed.  You can install it by typing:
sudo apt-get install ruby
-bash: ruby: command not found
```

To install ruby, simply run the *apt-get* command as instructed in the message:

```
sudo apt-get install ruby
```

The apt-get tool will display output listing any other packages required by Ruby (better known as *dependencies*):

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  libruby1.8 ruby1.8
Suggested packages:
  ruby1.8-examples rdoc1.8 ri1.8
The following NEW packages will be installed:
  libruby1.8 ruby ruby1.8
0 upgraded, 3 newly installed, 0 to remove and 135 not upgraded.
Need to get 1769kB of archives.
After unpacking 6267kB of additional disk space will be used.
Do you want to continue [Y/n]?
```

One the installation process is completed, you can verify that Ruby is installed by running it with the command line option to display the version information:

```
ruby -v
ruby 1.8.1 (2003-12-25) [i386-linux-gnu]
```
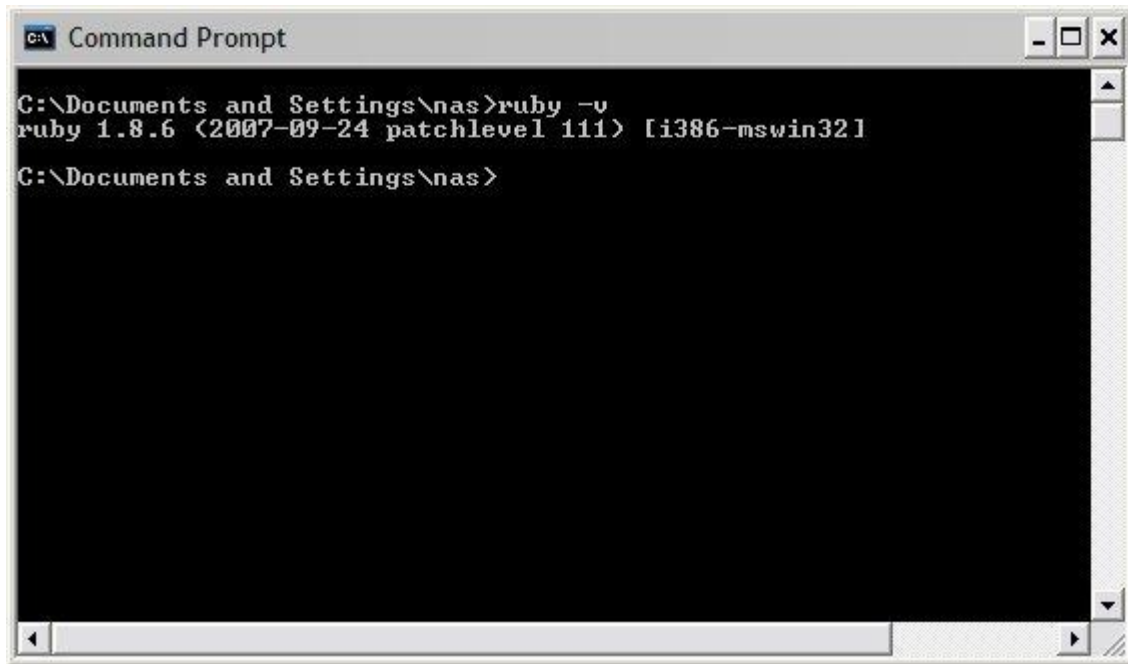
3.1.3 Ruby on Microsoft Windows

By far the easiest way to install Ruby on Windows is to use something called the *One-Click Ruby Installer*. This is an executable which, when run, performs the installation of Ruby onto a system. It also installs a mechanism by which Ruby may be easily removed from the system at a later date.

To use the *One-Click Ruby Installer* go to http://rubyforge.org, scroll down to the link to download the One-Click Installer and click on it. A second page will appear listing the various releases of the One-Click Installer. If you are feeling brave, choose the latest *Release Candidate*. If you prefer something a little more tried and tested, select the last non-release candidate download (for example, ruby186-26.exe).

Once the executable has downloaded, launch it as you would any other Windows application.

There are a number of ways to run Ruby once it is installed. The most basic approach is to start up a Windows Command Prompt and run Ruby from there. For example, we can output the version of Ruby installed on the system as follows:

```
Command Prompt                                          _ □ ×

C:\Documents and Settings\nas>ruby -v
ruby 1.8.6 (2007-09-24 patchlevel 111) [i386-mswin32]

C:\Documents and Settings\nas>
```

Another option is to launch the *fxri* tool from the Windows Start menu. This is an interactive tool which provides access to the Ruby documentation and also the Ruby console for interactively entering Ruby code:
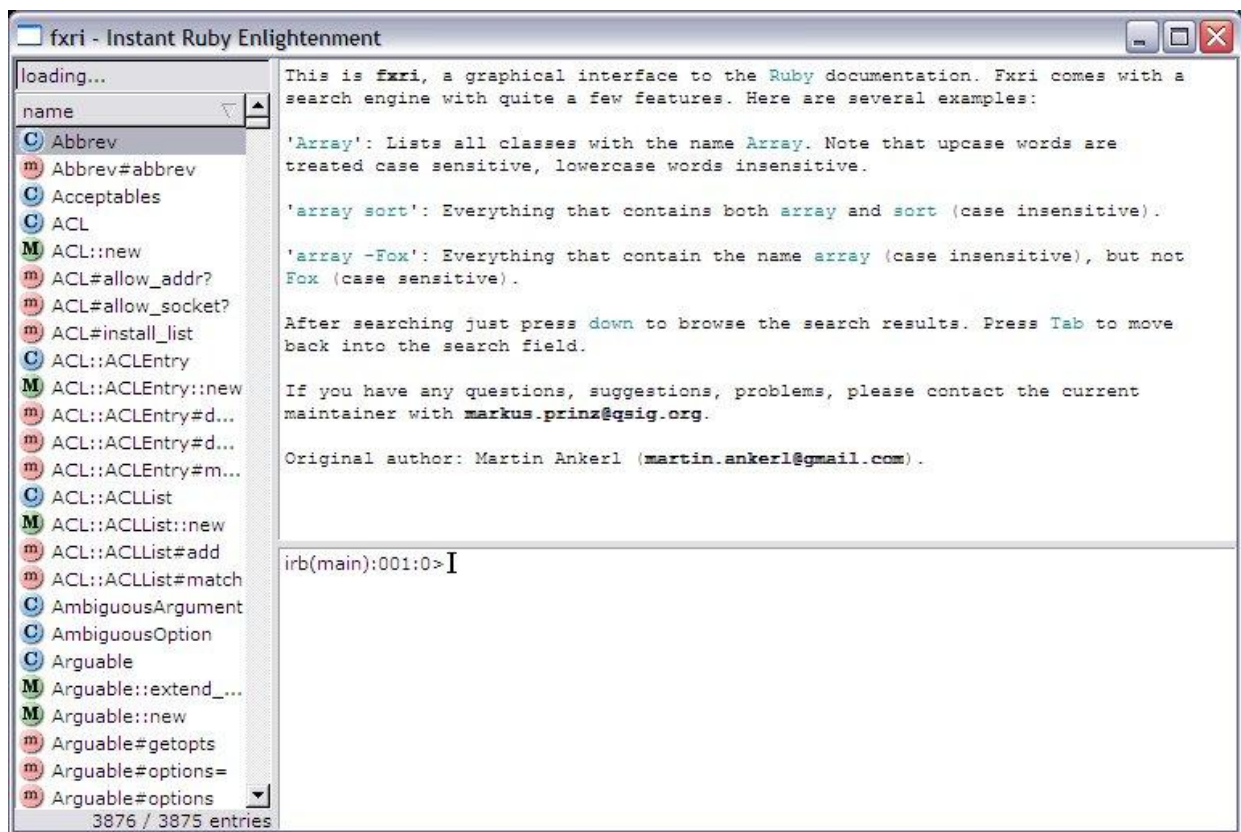
Once you have Ruby installed, it is time to create your first Ruby script which will be covered in the next chapter.

Chapter 4. Simple Ruby Examples

Not only is Ruby a flexible scripting language in terms of its syntax, it is also highly flexible in the ways in which scripts can be executed. In this chapter we will begin by looking at some simple Ruby examples, and then look at the variety of different ways Ruby code can be executed.

4.1 The Most Basic Ruby Example

Programming guides tend to follow the tradition of using "Hello World" as the first example of using a programming language. Typically such a program does nothing but output the string "Hello World" to the console window. Ruby Essentials is no exception to this rule, though we will modify it slightly to display "Hello Ruby!". Without further delay, let's get started:

On Linux or UNIX derived operating systems:

```
print "Hello Ruby!\n"
On Windows:
print "Hello Ruby!"
```

As you can see, all we need to do to output some text is enter a single line of Ruby code. You may recall if you have read the previous chapters that one of the strengths of Ruby is that it is fast and intuitive to learn. Clearly it would be hard to make printing a line of text any easier than a *print* statement followed by the text to be displayed (together with a newline character '\n' on Linux and UNIX).

As an example, let's compare this to the equivalent code in Java:

```
import java.io.*;

public class Hello {

        public static void main ( String[] args)
        {
                System.out.println ("Hello Ruby!\n");
        }
}
```

Hopefully, you are now beginning to see why Ruby is so popular. Programming languages such as Java require a significant amount of structure before even the simplest of tasks can be

performed. Before we go any further we need to learn how to execute our first Ruby example. This can be done in a number of ways, each of which will be covered in the following sections of this chapter.

4.2   Executing Ruby from the Command Line

Ruby allows lines of code to be executed as command line options to the *ruby* tool. This is achieved by using the '-e' command line flag. To execute our example 'Hello Ruby!' code, therefore, we could enter the following command:

```
ruby -e 'print "Hello Ruby!\n"'
Hello Ruby!
```

The '-e' flag only allows a single line of code to be executed, but that does not mean that multiple '-e' flags cannot be placed on a single command line to execute multiple lines:

```
ruby -e 'print "Hello Ruby!\n"' -e 'print "Goodbye Ruby!\n"'
Hello Ruby!
Goodbye Ruby!
```

4.3   Interactive Ruby Execution

In the previous chapter we discussed the fact that Ruby is an interpreted language. This essentially means that Ruby source code is compiled and executed at run time, rather than pre-compiled as is the case with languages such as C or C++. One of the advantages of being an interpreted language is that we can write Ruby code directly into the interpreter and have it executed interactively and in real-time. This is a great way to learn Ruby and to try out different code structures.

Interactive Ruby code is entered using the *irb* tool. If you are running Windows and installed Ruby using the one click installer, you already have *irb* installed. If you are running on Linux, there is a good chance *irb* is not yet installed. Verify the installation as follows:

```
irb -v
irb 0.9(02/07/03)
```

If you do not get the appropriate version information displayed, you will need to install *irb*. On Red Hat or Fedora Linux this can be achieved as follows:

```
su
yum install irb
```

On Debian, Ubuntu or other Debian derived Linux distributions use the apt-get tool:

```
sudo apt-get install irb
```

Once irb is installed, launch it as follows:

```
$ irb
irb(main):001:0>
```

Now, we can begin to execute Ruby code:

```
irb(main):001:0> puts 'Hello Ruby'
Hello Ruby
=> nil
irb(main):002:0>
```

We could also perform a calculation or two:

```
irb(main):002:0> 3 + 4
=> 7
irb(main):003:0> 8 * 7
=> 56
irb(main):004:0> 10 % 2
=> 0
```

As you can see, anything we type at the *irb* prompt gets executed as soon as we press the Enter key. Ruby truly is an interactive, interpreted language.

4.4 Executing Ruby from a File

Clearly the command line approach to execution is of limited use once you get beyond a few lines of Ruby script. A much more common approach is to place the Ruby script in a file, and then pass that file to the Ruby interpreter to run. To try this, create a file called *hello.rb* using your favorite editor and enter the following lines into it:

```
print "Hello Ruby!\n"
print "Goodbye Ruby!\n"
```

To execute this script, simply refer to it on the command line when launching *ruby*:

```
ruby hello.rb
Hello Ruby!
Goodbye Ruby!
```

4.5 Creating a Self Contained Ruby Executable on Linux or UNIX

Placing Ruby code into a file is obviously much easier and practical than using multiple **-e** command line options. Suppose, however, that we want to go one step further and be able to execute a Ruby based program simply by typing the name of the file containing the code, rather than prefixing it with the *ruby* command.

This can be achieved on Linux or UNIX by placing a special line at the top of the script file informing the environment responsible for executing the program (such as a Linux command shell) where to look for the Ruby interpreter. This special line consists of a '#', a '!' and the path to the *ruby* executable and is known affectionately as the *shebang*.

Firstly, you need to know where *ruby* is located on your system. Assuming it is already in your PATH environment variable you can use the *which* command to find it:

```
which ruby
/usr/bin/ruby
```

Given that *ruby* is in /usr/bin on the above system we can modify our sample application accordingly:

```
#!/usr/bin/ruby
print "Hello Ruby!\n"
print "Goodbye Ruby!\n"
```

We can now try running our *hello.rb* script:

```
./hello.rb
```

```
-bash: ./hello.rb: Permission denied
```

Clearly if you got the above output, there is a problem. This is simply a matter of the script not having *execute* permission. This can easily be rectified by using the *chmod* command to add execute permission to our script:

```
chmod 755 hello.rb
```

If we now try again we should get better results:

```
./hello.rb
Hello Ruby!
Goodbye Ruby!
```

4.6   Associating Ruby Files on Windows

The *Shebang* approach outlined in the preceding chapter does not work on Windows. The #! line will just be UNIX gibberish to a Windows system. The best way to configure a Windows system to detect that a file with a *.rb* file extension is to be launched with Ruby is to use Windows file type associations.

If you have used Windows extensively you will be familiar with the concept that it is possible to double click, for example, on a .doc file and have that file automatically loaded into Microsoft Word. This works because the system has been configured to associate .doc files with Word. Associating .rb files with Ruby is essentially the same thing.

First, it is important to note that if you installed Ruby on Windows using the One-Click Installer then .rb files will already have been associated with Ruby, so there is no need to perform the steps in this section. Simply type *hello.rb* at the command prompt and our example will run.

If you built Ruby yourself from the source code, or installed using a mechanism other than the One-Click Installer, you will need to associate .rb files with Ruby. The steps to achieve this are as follows:

Begin by checking whether the association is already configured:

```
C:\MyRuby>assoc .rb
File association not found for extension .rb
```

Assuming that the association is not already configured, take the following steps to complete the configuration:

```
C:\MyRuby>assoc .rb=rbFile
```

Check to see if the file type *rbfile* already exists:

```
C:\MyRuby>ftype rbfile
File type 'rbfile' not found or no open command associated with it.
```

Assuming it does not already exist (be sure to substitute the path to your Ruby installation in the following command):

```
C:\MyRuby>ftype rbfile="D:\Ruby\bin\ruby.exe" "%1" %*
```

Verify the setting:

```
C:\MyRuby>ftype rbfile
rbfile="D:\ruby\bin\ruby.exe" "%1" %*
```

Add .rb to the PATHEXT environment variable as follows:

```
C:\MyRuby>set PATHEXT=.rb;%PATHEXT%
```

Once the above settings are configured simply run the program by typing the filename at the command prompt (the .rb filename extension is not required):

```
C:\MyRuby> hello
Hello Ruby
```

The above steps can be placed in your Autoexec.bat file if you would like this association made every time you reboot your system.

Chapter 5. Commenting Ruby Code

Commenting code is a little like going to the dentist. We don't really want to have to do it but deep down we know it is a good thing to do. No matter how well written and self-explanatory your Ruby script it is still good practice to add comments. There a number of reasons for doing this. Firstly, it is possible that someone else may one day have to modify or fix bugs in your code. Good comments will help those who are new to your code to understand what it does. Secondly, no matter how well you understand your Ruby scripts now, I can assure you that in 6 months time you will look at the code and wonder how it all fits together (trust me, I've been programming for a long time and sometimes return to something I developed years ago and cannot believe I even wrote it!).

Comments are also useful for blocking out lines of code that you no longer wish to run (typically something that is done when trying to debug problems in complex programs).

5.1 What exactly is Commenting

Commenting is the process of marking content in a program such that it is ignored by the Ruby interpreter. This is typically used so that the programmer can write notes alongside the code describing what that code does such that other humans who look at the code will have a better chance of understanding the code.

Comments can span multiple lines, occupy a single line, or be tacked onto the end of a line of code.

5.2 Single Line Ruby Comments

Single line comments in a Ruby script are defined with the '#' character. For example, to add a single line comment to a simple script:

```
# This is a comment line - it explains that the next line of code displays a
welcome message
print "Welcome to Ruby!"
```

Comments lines can be grouped together:

```
# This is a comment line
# it explains that the next line of code displays
# a welcome message
```

Although a better way to implement multi-line comments is to use the comment begin and end markers described later in this chapter.

5.3   Comments on Lines of Code

It is common practice to place comments on the same line as the associated code to which the comment applies. For example, if we wanted to place a comment on the same line as our print statement we would do so by placing the comment after a '#' marker:

```
print "Welcome to Ruby!"      # prints the welcome message
```

Note that *everything* on the line after the '#' is ignored by the Ruby interpreter. You cannot, therefore, put more code after the '#' and expect it to be executed. Additional code must be placed on the next line.

5.4   Multi Line or Block Ruby Comments

Multiple lines of text or code can be defined as comments using the Ruby *=begin* and *=end* comment markers. These are known as the comment block markers.

For example, to provide a comment block containing multiple lines of descriptive text:

```
=begin
This is a comment line
it explains that the next line of code displays
a welcome message
=end
```

Similarly, lines of Ruby code can be blocked out so that they are not executed by the interpreter using the block markers:

```
=begin
print "Welcome to Ruby Essentials!"
print "Everything you need to know about Ruby"
=end
```

Chapter 6.   Understanding Ruby Variables

Variables are essentially a way to store a value and assign a name to that value for reference purposes. Variables take various forms ranging from integers to strings of characters. In this chapter we will take a look at how variables are declared and converted. We will also look at the much simpler area of Ruby constants.

6.1   Ruby Constants

A Ruby constant is used to store a value for the duration of a Ruby program's execution. Constants are declared by beginning the variable name with a capital letter (a common convention for declaring constants is to use uppercase letters for the entire name). For example:

```
MYCONSTANT = "hello"
=> "hello"
```

Unlike other programming languages, Ruby actually allows the value assigned to a constant to be changed after it has been created. The Ruby interpreter will, however, issue a warning - even though it allows the change:

```
MYCONSTANT = "hello2"
(irb):34: warning: already initialized constant Myconstant
=> "hello2"
```

6.2   Ruby and Variable Dynamic Typing

Many languages such as Java and C use what is known as *strong* or *static* variable typing. This means that when you declare a variable in your application code you must define the variable type. For example if the variable is required to store an integer value, you must declare the variable as an integer type. With such languages, when a variable has been declared as a particular type, the type cannot be changed.

Ruby, on the other hand, is a dynamically typed language. This has a couple of key advantages. Firstly it means that you do not need to declare a type when creating a variable. Instead, the Ruby interpreter looks at the type of value you are assigning to the variable and dynamically works out the variable type. Another advantage of this is that once a variable has been declared, you can dynamically change the variable type later in your code.

6.3 Declaring a Variable

Variables are declared and assigned values by placing the variable name and the value either side of the assignment operator (=). For example, to assign a value of 10 to a variable we will designate as "y" we would write the following:

```
y = 10
```

We have now created a variable called *y* and assigned it the value of 10.

In common with some other scripting languages, Ruby supports something called *parallel assignment*. This is useful if you need to assign values to a number of variables. One way to do this would be as follows:

```
a = 10
b = 20
c = 30
d = 40
```

The same result can be achieved more quickly, however, using parallel assignment:

```
a, b, c, d = 10, 20, 30, 40
```

6.4 Identifying a Ruby Variable Type

Once a Ruby variable has been declared it can often be helpful to find out the variable type. This can be achieved using the *kind_of?* method of the *Object* class. For example, to find out if our variable is an Integer we can use the *kind_of?* method as follows:

```
y.kind_of? Integer
=> true
```

We can also ask the variable exactly what class of variable it is using the *class* method:

```
y.class
=> Fixnum
```

This tells us that the variable is a fixed number class.

Similarly, we could perform the same task on a string variable we will call *s*:

```
s = "hello"
s.class
=> String
```

Here we see that the variable is of type String.

6.5   Changing Variable Type

One of simplest ways to change the type of a variable is to simply assign a new value to it. Ruby will dynamically change the type for that variable to match the type of the new value assigned. For example, we can create a variable containing an integer and verify the type:

```
x = 10
=> 10
x.class
=> Fixnum
```

Suppose, we now want to store a string in a variable named "x". All we need to do is make the assignment, and Ruby will change the variable type for us:

```
x = "hello"
=> "hello"
x.class
=> String
```

As we can see, the x variable is now a string.

6.6   Converting Variable Values

It is important to note that the above approach is a somewhat destructive way to change a variable type. Often something a little more subtle is needed. For example, we might want to read the value from a variable but convert the extracted value to a different type. The Ruby variable classes have methods that can be called to convert their value to a different type. For example, the *Fixnum* class has a method named *to_f* that can be used to retrieve the *integer* stored in a variable as *a floating point* value:

```
y = 20
```

```
=> 20
y.to_f
=> 20.0
```

Similarly, you can convert a Ruby integer to a string using the *to_s()* method. The *to_s()* method takes as an argument the number base to which you want the conversion made. If no number base is specified, decimal is assumed:

```
54321.to_s
=> "54321"
```

Alternatively, we could convert to binary by specifying a number base of 2:

```
54321.to_s(2)
=> "1101010000110001"
```

Or even to hexadecimal or octal:

```
54321.to_s(16)
=> "d431"
54321.to_s(8)
=> "152061"
```

In fact, we can use any number base between 1 and 36 when using the *to_s* method.

Chapter 7.   Ruby Variable Scope

Now that we have covered the basics of variables in Ruby the next task is to explain Ruby variable scope.

7.1   What is Variable Scope?

Scope defines where in a program a variable is accessible. Ruby has four types of variable scope, *local*, *global*, *instance* and *class*. In addition, Ruby has one constant type. Each variable type is declared by using a special character at the start of the variable name as outlined in the following table.

Name Begins With	Variable Scope
$	A global variable
@	An instance variable
[a-z] or _	A local variable
[A-Z]	A constant
@@	A class variable

In addition, Ruby has two *pseudo-variables* which cannot be assigned values. These are *nil* which is assigned to uninitialized variables and *self* which refers to the currently executing object. In the remainder of this chapter we will look at each of these variable scopes in turn.

7.2   Detecting the Scope of a Ruby Variable

Obviously, you can tell the scope of a variable by looking at the name. Sometimes, however, you need to find out the scope programmatically. A useful technique to find out the scope of a variable is to use the *defined?* method. *defined?* returns the scope of the variable referenced, or *nil* if the variable is not defined in the current context:

```
x = 10
=> 10
defined? x
=> "local-variable"


$x = 10
=> 10
defined? $x
=> "global-variable"
```

7.3   Ruby Local Variables

Local variables are local to the code construct in which they are declared. For example, a local variable declared in a method or within a loop cannot be accessed outside of that loop or method. Local variable names must begin with either an underscore or a lower case letter. For example:

```
loopcounter = 10
_LoopCounter = 20
```

7.4   Ruby Global Variables

Global variables in Ruby are accessible from anywhere in the Ruby program, regardless of where they are declared. Global variable names are prefixed with a dollar sign ($). For example:

```
$welcome = "Welcome to Ruby Essentials"
```

Use of global variables is strongly discouraged. The problem with global variables is that, not only are they visible anywhere in the code for a program, they can also be changed from anywhere in the application. This can make tracking bugs difficult.

It is useful to know, however, that a number of pre-defined global variables are available to you as a Ruby developer to obtain information about the Ruby environment. A brief summary of each of these variables is contained in the following table.

Variable Name	Variable Value
`$@`	The location of latest error
`$_`	The string last read by `gets`
`$.`	The line *number* last read by interpreter
`$&`	The string last matched by regexp
`$~`	The last regexp match, as an array of subexpressions
`$n`	The *nth* subexpression in the last match (same as `$~[n]`)
`$=`	The case-insensitivity flag
`$/`	The input record separator
`$\`	The output record separator
`$0`	The name of the ruby script file currently executing
`$*`	The command line arguments used to invoke the script
`$$`	The Ruby interpreter's process ID

`$?`	The exit status of last executed child process

For example we can execute the *gets* method to take input from the keyboard, and then reference the *$_* variable to retrieve the value entered:

```
irb(main):005:0> gets
hello
=> "hello\n"
irb(main):006:0> $_
=> "hello\n"
```

Alternatively we could find the process ID of the Ruby interpreter:

```
irb(main):007:0> $$
=> 17403
```

7.5   Ruby Class Variables

A class variable is a variable that is shared amongst all instances of a class. This means that only one variable value exists for all objects instantiated from this class. This means that if one object instance changes the value of the variable, that new value will essentially change for all other object instances.

Another way of thinking of thinking of class variables is as global variables within the context of a single class.

Class variables are declared by prefixing the variable name with two @ characters (@@). Class variables must be initialized at creation time. For example:

```
@@total = 0
```

7.6   Ruby Instance Variables

Instance variables are similar to Class variables except that their values are local to specific instances of an object. For example if a class contains an instance variable called *@total*, if one instance of the object changes the current value of *@total* the change is local to only the object that made the change. Other objects of the same class have their own local copies of the variable which are independent of changes made in any other objects.

Instance variables are declared in Ruby by prefixing the variable name with a single @ sign:

```
@total = 10
```

7.7 Ruby Constant Scope

Ruby constants are values which, once assigned a value, should not be changed. I say *should* because Ruby differs from most programming languages in that it allows a constant value to be changed after it has been declared, although the interpreter will protest slightly with a warning message.

Constants declared within a class or module are available anywhere within the context of that class or module. Constants declared outside of a class or module are assigned global scope.

Chapter 8. Ruby Number Classes and Conversions

Just about everything in Ruby is an object. Perhaps one of most surprising things is that even numbers are objects in Ruby. Most other programming languages treat numbers as primitives (which essentially means they are the building blocks from which objects are built). This means that each number type is associated with a number class and can be manipulated using the methods of that class.

8.1 Ruby Number Classes

Ruby provides a number of built-in number classes. In this section we will explore some of the more commonly used classes.

8.1.1 Integer Class

The base class from which the following number classes are derived.

8.1.2 Fixnum Class

A Fixnum holds Integer values that can be represented in a native machine word (minus 1 bit). This effectively means that the maximum range of a Fixnum value depends on the architecture of the system on which the code is executing.

If an operation performed on a Fixnum exceeds the range defined by the system's machine word, the value is automatically converted by the interpreter to a Bignum.

8.1.3 Bignum Class

Bignum objects hold integers that fall outside the range of the Ruby Fixnum class. When a calculation involving Bignum objects returns a result that will fit in a Fixnum, the result is converted to Fixnum.

8.1.4 Float Class

The Float object represents real numbers based on the native architecture's double-precision floating point representation.

8.1.5 Rational Class

Rational implements a rational class for numbers.

A rational number is a number that can be expressed as a fraction p/q where p and q are integers and q is not equal to zero. A rational number p/q is said to have numerator p and denominator q. Numbers that are not rational are called irrational numbers.

8.2 Converting Numbers in Ruby

Numbers can be converted from one type to another using the Ruby *Integer* and *Float* methods. These methods take as an argument the value to be converted. For example:

8.2.1 Convert Floating Point Number to an Integer

```
Integer (10.898)
=> 10
```

8.2.2 Convert a String to an Integer

```
Integer ("10898")
=> 10898
```

8.2.3 Convert a Hexadecimal Number to an Integer

```
Integer (0xA4F5D)
=> 675677
```

8.2.4 Convert an Octal Number to an Integer

```
Integer (01231)
=> 665
```

8.2.5 Convert a Binary Number to an Integer

```
Integer (0b01110101)
=> 117
```

8.2.6 Convert a Character to the ASCII Character Code

For Ruby version 1.8 or earlier:

```
Integer (?e)
=> 101
```

For Ruby version 1.9 or later:

```
"e".getbyte(0)               #ASCII to Integer
=> 101
```

Similarly, we can perform conversions to floating point using the Float method:

8.2.7   Convert an Integer Floating Point

```
Float (10)
=> 10.0
```

8.2.8   Convert a String to Floating Point

```
Float ("10.09889")
=> 10.09889
```

8.2.9   Convert a Hexadecimal Number to Floating Point

```
Float (0xA4F5D)
=> 675677.0
```

8.2.10 Convert an Octal Number to a Floating Point

```
Float (01231)
=> 665.0
```

8.2.11 Convert a Binary Number to Floating Point

```
Float (0b01110101)
=> 117
```

8.2.12 Convert a Character to a Floating Point ASCII Character Code

For Ruby version 1.8 or earlier:

```
Float (?e)
=> 101.0
```

For Ruby version 1.9 or later:

```
"e".getbyte(0)
=> 101.0
```

Chapter 9. Ruby Methods

Ruby methods provide a way to organize code and promote re-use. Rather than create long sections of Ruby code, the code is instead organized into logical groups that can be called when needed and re-used without having to re-write the same code over and over. Methods are simple to use, in fact you only need to do two things with a method, declare it and call it.

9.1 Declaring and Calling a Ruby Method

The syntax of a Ruby method is as follows:

```
def name( arg1, arg2, arg3, ... )
   .. ruby code ..
   return value
end
```

The *name* specifies how we will refer to this method when we call it. The *args* specify values that are passed through to the method to be processed. The *ruby code* section represents the body of the function that performs the processing. The optional *return* statement allows a *value* to be returned to the section of code which called the method (for example to return a status or the result of a calculation).

The following simple example shows a method defined and called. All the method does is display a string:

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

saysomething
```

9.2 Passing Arguments to a Method

The above example did not pass any arguments through to the function. Commonly a function is designed to perform some task on a number of arguments as in the following example:

```
def multiply(val1, val2 )
    result = val1 * val2
    puts result
```

```
end

multiply( 2, 10 )
multiply( 4, 20 )
multiply( 10, 40 )
multiply( 6, 7 )
```

In this example, the method is called multiple times, passing through arguments that are then used in the method to perform a calculation on the arguments, displaying the result. To achieve this without methods it would be necessary to repeat the code in the method 4 times over. Clearly, placing the code in a method and re-using it over and over is a much more efficient approach.

Next we need to look at how a method might return a value.

9.3   Passing a Variable Number of Arguments to a Method

In the previous section of this chapter we looked at specifying a fixed number of arguments accepted by a method. Sometimes we don't always know in advance how many arguments will be needed. Ruby addresses this by allowing a method to be declared with a variable number of arguments. This achieved by using *args* when declaring the method. The arguments passed to the method are then placed in an array where they may be accessed in the body of the method (see the _Understanding Ruby Arrays_ for details on using arrays):

```
irb(main):062:0> def displaystrings( *args )
irb(main):063:1>         args.each {|string| puts string}
irb(main):064:1> end
=>nil

displaystrings("Red")
Red

displaystrings("Red", "Green")
Red
Green

irb(main):067:0> displaystrings("Red", "Green", "Blue")
Red
```

```
Green
Blue
```

As you can see, the method can handle any number of arguments passed through.

9.4 Returning a Value from a Function

The *return* statement is used to return a value from a method and the assignment (=) method is used to accept that *return* value at the point that the method is called.

As an example, we will declare a method which multiplies two arguments and returns the result:

```
def multiply(val1, val2 )
     result = val1 * val2
     return result
end


value = multiply( 10, 20 )
puts value
```

The above example passes 10 and 20 through to the *multiply* method. The method multiplies these two values and returns the result. The assignment method (=) assigns the result to the variable *value* which is then displayed using *puts*.

It is important to note that a method can return one, and only one value or object. If you need to return multiple values, consider placing the results in an array and returning the array.

9.5 Ruby Method Aliases

Ruby allows a method to be aliased, thereby creating a copy of a method with a different name (although invoking the method with either name ultimately calls the same object). For example:

```
irb(main):001:0> def multiply(val1, val2 )
irb(main):002:1>       result = val1 * val2
irb(main):003:1>       return result
irb(main):004:1> end
=> nil
```

```
alias docalc multiply
=> nil

docalc( 10, 20 )
=> 200

multiply( 10, 20 )
=> 200
```

Chapter 10.   Ruby Ranges

Ruby Ranges allow data to be represented in the form of a range (in other words a data set with start and end values and a logical sequence of values in between). The values in a range can be numbers, characters, strings or objects. In this chapter we will look at the three types of range supported by Ruby, namely sequences, conditions and intervals.

10.1 Ruby Sequence Ranges

Sequence ranges in Ruby are used to create a range of successive values - consisting of a start value, an end value and a range of values in between.

Two operators are available for creating such ranges, the inclusive two-dot (..) operator and the exclusive three-dot operator (...). The inclusive operator includes both the first and last values in the range. The exclusive range operator excludes the last value from the sequence. For example:

```
1..10    # Creates a range from 1 to 10 inclusive
1...10   # Creates a range from 1 to 9
```

A range may be converted to an array using the Ruby *to_a* method. For example:

```
(1..10).to_a
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

(1...10).to_a
=> [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

As mentioned previously, ranges are not restricted to numerical values. We can also create a character based range:

```
('a'..'l').to_a
=> ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l"]
```

Also, we can create ranges based on string values:

```
('cab'..'car').to_a
```

```
=> ["cab", "cac", "cad", "cae", "caf", "cag", "cah", "cai", "caj", "cak",
"cal", "cam",
"can", "cao", "cap", "caq", "car"]
```

Range values may also be objects. The only requirements for placing an object in a range is that the object provides a mechanism for returning the next object in the sequence via *succ* and it must be possible to compare the objects using the <=> operator.

10.2 Using Range Methods

Given the object-oriented nature of Ruby it should come as no surprise to you that a range is actually an object. As such, there are a number of methods of the Range class which can be accessed when working with a range object.

```
words = 'cab'..'car'

words.min        # get lowest value in range
=> "cab"

words.max        # get highest value in range
=> "car"

words.include?('can') # check to see if a value exists in the range
=> true

words.reject {|subrange| subrange < 'cal'} # reject values below a specified
range value
=> ["cal", "cam", "can", "cao", "cap", "caq", "car"]

words.each {|word| puts "Hello " + word} # iterate through each value and
perform a task
Hello cab
Hello cac
Hello cad
Hello cae
Hello caf
Hello cag
Hello cah
```

```
Hello cai

Hello caj

Hello cak

Hello cal

Hello cam

Hello can

Hello cao

Hello cap

Hello caq

Hello car
```

10.3 Ruby Ranges as Conditional Expressions

Ruby Ranges can also be used as conditional expressions in looping conditions. The range start value represents the start of the loop, which runs until the range end marker is detected.

```
while input = gets
puts input + " triggered" if input =~ /start/ .. input =~ /end/
end
```

10.4 Ruby Range Intervals

Ranges are ideal for identifying if a value falls within a particular range. For example, we might want to know whether a number is within a certain range, or a character within a certain group of letters arranged in alphabetical order. This is achieved using the === equality operator:

```
(1..20) === 15        # Does the number fit in the range 1 to 20
=> true
('k'..'z') === 'm'    # Does the letter fall between the letters 'k' and 'z'
in the alphabet?
=> true
```

10.5 Ranges in case Statements

Ranges are perhaps at their most powerful when they are used in conjunction with a *case statement* (for more information also see *The Ruby case Statement*):

```
score = 70

result = case score
    when 0..40 then "Fail"
    when 41..60 then "Pass"
    when 61..70 then "Pass with Merit"
    when 71..100 then "Pass with Distinction"
    else "Invalid Score"
end

puts result
```

Chapter 11.   Understanding Ruby Arrays

In *Understanding Ruby Variables* we looked at storing data (such as numbers, strings and boolean true or false values) in memory locations known as variables. The variable types covered in those chapters were useful for storing one value per variable. Often, however, it is necessary to group together multiple variables into a self contained object. This is where Ruby arrays come in. The objective of this chapter, therefore, is to introduce the concept of Arrays in Ruby and provide an overview of the creation and manipulation of such objects. In the next chapter (*Advanced Ruby Arrays*) we will look at more ways to work with arrays.

11.1 What is a Ruby Array

A Ruby array is an object that contains a number of items. Those items can be variables (such as String, Integer, Fixnum Hash etc) or even other objects (including other arrays to make a multidimensional array). Once you have grouped all the items into the array you can then perform tasks like sorting the array items into alphabetical or numerical order, accessing and changing the value assigned to each array item, and passing the group of items as an argument to a Ruby function.

11.2 How to Create a Ruby Array

Ruby provides a number of mechanisms for creating an array. Arrays essentially involve the use of the Ruby *Array* class. We could, therefore, create an uninitialized array using the *new* method of the *Array* class:

```
days_of_week = Array.new
```

We now have an array called *days_of_week* with nothing in it. In fact, we can verify if an array is empty by calling the *empty?* method of the Ruby *Array* class which will return *true* if the array is empty:

```
days_of_week.empty?
=> true
```

We can also initialize an array with a preset number of elements by passing through the array size as an argument to the *new* method:

```
days_of_week = Array.new(7)
```

```
=> [nil, nil, nil, nil, nil, nil, nil]
```

Note that when we preset the size, all the elements are initialized to *nil*. Now we need some way of populating an array with elements.

11.3 Populating an Array with Data

Having created an array we need to add some data to it. One way to do this is to place the same data in each element during the created process:

```
days_of_week = Array.new(7, "today")
=> ["today", "today", "today", "today", "today", "today", "today"]
```

Another option is to use the [] method of the *Array* class to specify the elements one by one:

```
days_of_week = Array[ "Mon", "Tues", "Wed", "Thu", "Fri", "Sat", "Sun" ]
=> ["Mon", "Tues", "Wed", "Thu", "Fri", "Sat", "Sun"]
```

This can also be abbreviated to just the array name and the square brackets:

```
days_of_week = [ "Mon", "Tues", "Wed", "Thu", "Fri", "Sat",
"Sun" ]
```

This will not only create the array for us, but also populate it with the element data.

11.4 Finding Out Information about a Ruby Array

Once an array exists, it can be useful to find out information about that array and its elements. As we mentioned earlier, it is possible to find out if an array is empty or not:

```
days_of_week.empty?
=> true
```

We can also find out the size of an array using the *size* method of the Array class:

```
days_of_week = Array.new(7)
days_of_week.size
=> 7
```

It is also possible find out what type of object is contained in array by using the *array index* value of the element we want to interrogate combined with the *class* method:

```
days_of_week = [ "Mon", 15, "Wed", 16, "Thu", 17 ]

days_of_week[0].class
=> String

days_of_week[1].class
=> Fixnum
```

11.5 Accessing Array Elements

The elements of an array can be accessed by referencing the index of the element in question in conjunction with the [] method. To access the first and second elements of our array therefore:

```
days_of_week[0]
=> "Mon"

days_of_week[1]
=> "Tues"
```

The Array class *at* method can be used to similar effect:

```
days_of_week.at(0)
=> "Mon"
```

The last element of an array may be accessed by specifying an array index of -1. For example:

```
days_of_week[-1]
=> "Sun"
```

The first and last elements may be accessed by using the *first* and *last* methods of the Array class:

```
days_of_week.first
```

```
=> "Mon"


days_of_week.last
=> "Sun"
```

11.6 Finding the Index of an Element

Often when working with arrays it is necessary to find out the index of a particular element. This can be achieved using the *index* method which returns the index of the first element to match the specified criteria. For example, to identify the index value of the "Wed" element of our days of the week array:

```
days_of_week.index("Wed")
=> 2
```

The *rindex* method can be used to find the last matching element in an array.

A subset of an array's elements may be extracted by specifying the start point and the number of elements to extract. For example, to start at element 1 and read 3 elements:

```
days_of_week[1, 3]
=> ["Tues", "Wed", "Thu"]
```

Similarly, we can specify a range (for information on ranges see the *Ruby Ranges*).

```
days_of_week[1..3]
=> ["Tues", "Wed", "Thu"]
```

Alternatively, the *slice* method of the Array class may be used:

```
 days_of_week.slice(1..3)
=> ["Tues", "Wed", "Thu"]
```

Chapter 12.   Advanced Ruby Arrays

In the *Understanding Ruby Arrays* we looked at the basics of arrays in Ruby. In this chapter we will look at arrays in more detail.

12.1 Combining Ruby Arrays

Arrays in Ruby can be concatenated using a number of different approaches. One option is to simply *add,* them together:

```
days1 = ["Mon", "Tue", "Wed"]
days2 = ["Thu", "Fri", "Sat", "Sun"]
days = days1 + days2
=> ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
```

Alternatively, the *concat* method may be called to achieve the same result:

```
days1 = ["Mon", "Tue", "Wed"]
days2 = ["Thu", "Fri", "Sat", "Sun"]
days = days1.concat(days2)
```

Elements may also be appended to an existing array using the << method. For example:

```
days1 = ["Mon", "Tue", "Wed"]
days1 << "Thu" << "Fri" << "Sat" << "Sun"
=> ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
```

12.2 Intersection, Union and Difference

Ruby's support for array manipulation goes beyond that offered by many other scripting languages. One area where this is particularly true involves the ability to create new arrays based on the union, intersection and difference of two arrays. These features are provided via the following *set operation* symbols:

Operator	Description
-	Difference - Returns a new array that is a copy of the first array with any items that also appear in second array removed.
&	Intersection - Creates a new array from two existing arrays containing only

		elements that are common to both arrays. Duplicates are removed.
	|	Union - Concatenates two arrays. Duplicates are removed.

A few examples will help to clarify these operations. Let's begin by creating two arrays:

```
operating_systems = ["Fedora", "SuSE", "RHEL", "Windows", "MacOS"]


linux_systems = ["RHEL", "SuSE", "PCLinuxOS", "Ubuntu", "Fedora"]
```

Now, we can create a union of the two arrays:

```
operating_systems | linux_systems
=> ["Fedora", "SuSE", "RHEL", "Windows", "MacOS", "PCLinuxOS", "Ubuntu"]
```

As we can see from the above output, the union operation joined one array to another, but removed any duplicate array elements.

Next we can perform an intersection:

```
operating_systems & linux_systems
=> ["Fedora", "SuSE", "RHEL"]
```

This time, we only get elements that are common to both arrays.

Finally, we can try a "difference" operation:

```
operating_systems - linux_systems
=> ["Windows", "MacOS"]
```

In this case, the new array provides us with the difference between the two arrays. In other words we get a new array that contains items which are present in *operating systems*, but *not* present in *linux_systems*. It is important to be clear on the point that we are not simply removing duplicate entries with this operator, but rather removing items from the array specified by the left hand operand that are also present in the array specified by the right hand operand. This can be demonstrated by the fact that switching the operands gives us very different results:

```
linux_systems - operating_systems
=> ["PCLinuxOS", "Ubuntu"]
```

12.3 Identifying Unique Array Elements

The *uniq* method of the Array class can be used to remove duplicated elements from an array. For example:

```
linux_systems = ["RHEL", "SuSE", "PCLinuxOS", "Ubuntu", "Fedora", "RHEL",
"SuSE"]

linux_systems.uniq
=> ["RHEL", "SuSE", "PCLinuxOS", "Ubuntu", "Fedora"]
```

Note that in this instance, the original array is unchanged by the *uniq* method. If you really wanted to strip out the duplicates from an array such that the array itself is altered, use the *uniq!* method as follows:

```
linux_systems
=> ["RHEL", "SuSE", "PCLinuxOS", "Ubuntu", "Fedora", "RHEL", "SuSE"]

linux_systems.uniq!
=> ["RHEL", "SuSE", "PCLinuxOS", "Ubuntu", "Fedora"]

linux_systems
=> ["RHEL", "SuSE", "PCLinuxOS", "Ubuntu", "Fedora"]
```

12.4 Pushing and Popping Array Elements

An array in Ruby can be treated as a Last In First Out (LIFO) stack where items are *pushed* onto and *popped* off the array. This is achieved, unsurprisingly, using *push* and *pop* methods.

For example we can create an array and then push elements onto it:

```
colors = ["red", "green", "blue"]
=> ["red", "green", "blue"]
colors.push "indigo"
```

```
=> ["red", "green", "blue", "indigo"]


colors.push "violet"
=> ["red", "green", "blue", "indigo", "violet"]
```

Using the *pop* method we can also pop elements from the array:

```
colors.pop
=> "violet"


colors.pop
=> "indigo"
```

Note how the elements are popped out of the array, starting with the last to be pushed onto it. Hence the term *Last In First Out* (LIFO).

12.5 Ruby Array Comparisons

Ruby arrays may be compared using the ==, <=> and *eql?* methods.

The == method returns *true* if two arrays contain the same number of elements and the same contents for each corresponding element.

The *eql?* method is similar to the == method with the exception that the values in corresponding elements are of the same value type.

Finally, the <=> method (also known as the "spaceship" method) compares two arrays and returns 0 if the arrays are equal, -1 one if the elements are less than those in the other array, and 1 if they are greater.

12.6 Modifying Arrays

A new element may be inserted into an array using the *insert* method. This method takes as arguments the index value of the element to be inserted, followed by the new value. For example, to insert a new color between the red and green elements:

```
colors = ["red", "green", "blue"]
=> ["red", "green", "blue"]


colors.insert( 1, "orange" )
```

```
=> ["red", "orange", "green", "blue"]
```

Any array element may be changed simply by assigning a new value using the array element index:

```
colors = ["red", "green", "blue"]
=> ["red", "green", "blue"]=> ["red", "yellow", "blue"]

colors[1] = "yellow"
=> "yellow"

colors
=> ["red", "yellow", "blue"]
```

Multiple elements can be changed by making use of a range:

```
colors = ["red", "green", "blue"]
=> ["red", "green", "blue"]

colors[1..2] = "orange", "pink"
=> ["orange", "pink"]

colors
=> ["red", "orange", "pink"]
```

12.7 Deleting Array Elements

It is not uncommon to need to delete individual elements from an array. A deletion can be performed either based on the content of an array element, or on the index position.

To delete based on an index use the *delete_at* method:

```
colors = ["red", "green", "blue"]
=> ["red", "green", "blue"]

colors.delete_at(1)
=> "green"
```

```
colors
=> ["red", "blue"]
```

To delete array elements based on content use the *delete* method:

```
colors = ["red", "green", "blue"]
=> ["red", "green", "blue"]

colors.delete("red")
=> "red"

colors
=> ["green", "blue"]
```

12.8 Sorting Arrays

Arrays are sorted in Ruby using the *sort* and *reverse* methods:

```
numbers = [1, 4, 6, 7, 3, 2, 5]
=> [1, 4, 6, 7, 3, 2, 5]

numbers.sort
=> [1, 2, 3, 4, 5, 6, 7]
```

As with the *uniq* method, the sorting can be applied to the actual array with the *sort!* method.

The order of the elements in an array can be reversed using the *reverse* method:

```
numbers = [1, 4, 6, 7, 3, 2, 5]
=> [1, 4, 6, 7, 3, 2, 5]

numbers.sort!
=> [1, 2, 3, 4, 5, 6, 7]

numbers.reverse
```

```
=> [7, 6, 5, 4, 3, 2, 1]
```

Chapter 13.   Ruby Operators

In this chapter we will cover the basics of using operators in Ruby to build expressions. Ruby provides a number of different types of operators including:

- Assignment Operators
- Math Operators
- Comparison Operators
- Bitwise Operators

13.1 The Anatomy of a Ruby Operation

In Ruby, as with most other programming languages, operations consist of values on which the calculation is to be performed (called *operands*) and an *operator* which dictates the operation to be performed. Typically, the *operands* are placed either side of the *operator*. Optionally, the *assignment* operator (=) can be used to assign the result of the operation to, for example, a variable. Let's take the most basic of operations, executed in *irb*:

```
irb(main):036:0> 1 + 1
=> 2
```

Now let's assign the result to a variable called *result*:

```
irb(main):037:0> result = 1 + 1
=> 2
```

13.2 Performing Ruby Arithmetic using Operators

Ruby provides a number of basic operators for performing arithmetic. These are as follows:

Operator	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
**	Exponent - Performs exponential (power) calculation on operators

Note that when performing division, if you do not want the result to be truncated to a round number, one of the operands in the operation must be expressed as a float:

```
irb(main):001:0> 10 / 7
=> 1
```

In the above example the result has been rounded down to the nearest whole number. On the other hand, if we express one of the operands as a float, we get a more accurate answer:

```
irb(main):003:0> 10.0 / 7
=> 1.42857142857143
```

13.3 Ruby Assignment Operators

Earlier in this chapter we looked the basic assignment operator (=) which allows us to assign the result of an expression, for example *y = 10*.

The '=' assignment operator does not make any changes to the value before it is assigned to the variable. A number of assignment operators are available, however, that perform arithmetic on the value before assigning it to the variable. These are essentially combined arithmetic and assignment operators. The most common operators of this type, and their "long hand" equivalents are shown below:

Combined Operator	Equivalent
x += y	x = x + y
x -= y	x = x - y
x /= y	x = x / y
x *= y	x = x * y
x %= y	x = x % y
x **= y	x = x ** y

These combined operators provide a short way of assigning the results of arithmetic expressions between two variables or a variable and a value and having the result assigned to the first variable. For example:

```
x = 10
x += 5 # Assigns a value of 15 to variable x (the same as x = x + 5)
```

```
y = 20
y -= 10  # Assigns a value of 10 to variable y (the same as y = y - 10)


x = 10;
y = 5;


x /= y; # Assigns a value of 2 to variable x (the same as x = x / y)
```

13.4 Parallel Assignment

Ruby also supports the parallel assignment of variables. This enables multiple variables to be initialized with a single line of Ruby code. For example:

```
a = 10
b = 20
c = 30
```

The above code may be more quickly declared using parallel assignment:

```
a, b, c = 10, 20, 30
```

Parallel assignment is also useful for swapping the values held in two variables:

```
a, b = b, c
```

13.5 Ruby Comparison Operators

Comparison operators are used to check for equality between two values. Ruby provides a number of such operators:

Comparison Operator	Description
==	Tests for equality. Returns *true* or *false*
.eql?	Same as ==.
!=	Tests for inequality. Returns *true* for inequality or *false* for equality
<	Less than. Returns *true* if first operand is less than second operand. Otherwise returns *false*

>	Greater than. Returns *true* if first operand is greater than second operand. Otherwise returns *false*.
>=	Greater than or equal to. Returns *true* if first operand is greater than or equal to second operand. Otherwise returns *false*.
<=	Less than or equal to. Returns *true* if first operand is less than or equal to second operand. Otherwise returns *false*.
<=>	Combined comparison operator. Returns 0 if first operand equals second, 1 if first operand is greater than the second and -1 if first operand is less than the second.

Let's look at some of these operators in action:

```
irb(main):006:0> 1 == 1
=> true

irb(main):015:0> 1.eql? 2
=> false

irb(main):016:0> 10 < 1
=> false

irb(main):019:0> 10 <=> 10
=> 0

irb(main):020:0> 10 <=> 9
=> 1

irb(main):021:0> 10 <=> 11
=> -1
```

13.6 **Ruby Bitwise Operators**

Bitwise operators allow operations to be performed on number at the bit level. As you are probably already aware, computers deal solely with binary (in other words ones and zeros). For example, the computer sees the number 520 as 01010.

The Ruby bitwise operators allow us to operate at the level of the ones and zeros that make up a number:

Combined Operator	Equivalent
~	Bitwise NOT (Complement)
|	Bitwise OR
&	Bitwise AND
^	Bitwise Exclusive OR
<<	Bitwise Shift Left
>>	Bitwise Shift Right

As with the math operators, Ruby also provides a number of combined bitwise operators (for example ~=, >>=, <<= ^=, &=).

13.7 Summary

In this chapter of Ruby Essentials we have covered the basics of Ruby operators, expressions and assignments. In the next chapter we will look at the all important area of Ruby Operator Precedence.

Chapter 14.   Ruby Operator Precedence

In the previous chapter of Ruby Essentials we looked at Ruby operators and expressions. An equally important area to understand is operator precedence. This is essentially the order in which the Ruby interpreter evaluates expressions comprising more than one operator.

14.1 An Example of Ruby Operator Precedence

When humans evaluate expressions, they usually do so starting at the left of the expression and working towards the right. For example, working from left to right we get a result of 300 from the following expression:

```
10 + 20 * 10 = 300
```

This is because we, as humans, add 10 to 20, resulting in 30 and then multiply that by 10 to arrive at 300. Ask Ruby to perform the same calculation and you get a very different answer:

```
irb(main):003:0> 10 + 20 * 10
=> 210
```

This is a direct result of *operator precedence*. Ruby has a set of rules that tell it in which order operators should be evaluated in an expression. Clearly, Ruby considers the multiplication operator (*) to be of a higher precedence than the addition (+) operator.

14.2 Overriding Operator Precedence

The precedence built into Ruby can be overridden by surrounding the lower priority section of an expression with parentheses. For example:

```
(10 + 20) * 10
=> 300
```

In the above example, the expression fragment enclosed in parentheses is evaluated before the higher precedence multiplication.

14.3 Operator Precedence Table

The following table provides a reference when you need to know the operator precedence used by Ruby. The table lists all operators from highest precedence to lowest.

Method	Operator	Description
Yes	`[ ] [ ]=`	Element reference, element set
Yes	`**`	Exponentiation (raise to the power)
Yes	`! ~ + -`	Not, complement, unary plus and minus (method names for the last two are `+@` and `-@`)
Yes	`* / %`	Multiply, divide, and modulo
Yes	`+ -`	Addition and subtraction
Yes	`>> <<`	Right and left bitwise shift
Yes	`&`	Bitwise `AND'
Yes	`^ |`	Bitwise exclusive `OR' and regular `OR'
Yes	`<= < > >=`	Comparison operators
Yes	`<=> == === != =~ !~`	Equality and pattern match operators (`!=` and `!~` may not be defined as methods)
	`&&`	Logical `AND'
	`||`	Logical `AND'
	`.. ...`	Range (inclusive and exclusive)
	`? :`	Ternary if-then-else
	`= %= { /= -= += |= &= >>= <<= *= &&= ||= **=`	Assignment
	`defined?`	Check if specified symbol defined
	`not`	Logical negation
	`or and`	Logical composition
	`if unless while until`	Expression modifiers
	`begin/end`	Block expression

Operators with a *Yes* in the method column are actually methods, and as such may be overridden.

Chapter 15.   Ruby Math Functions and Methods

The Ruby *Math* module provides the Ruby programmer with an extensive range of methods for performing mathematical tasks. In addition, the *Math* module includes two commonly used mathematical constants.

15.1 Ruby Math Constants

The Ruby *Math* module includes common math constants. A list of constants may be accessed using the *constants* method:

```
Math.constants
=> ["E", "PI"]
```

As we can see, as of the current version of Ruby, only two constants are defined. We can access these using :: notation:

```
Math::PI
=> 3.14159265358979


Math::E
=> 2.71828182845905
```

15.2 Ruby Math Methods

As mentioned previously, Ruby provides an extensive range of math related methods. These are listed and described in the following table.

Method name	Description
Math.acos, Math.acos!	Arc cosine
Math.acosh, Math.acosh!	Hyperbolic arc cosine
Math.asin, Math.asin!	Arc sine
Math.asinh, Math.asinh	Hyperbolic arc sine
Math.atan, Math.atan!, Math.atan2, Math.atan2!	Arc tangent. atan takes an x argument. atan2 takes x and y arguments
Math.atanh, Math.atanh!	Hyperbolic arc tangent
Math.cos, Math.cos!	Cosine
Math.cosh, Math.cosh	Hyperbolic cosine

Math.sin, Math.sin!	Sine
Math.erf	Error function
Match.erfc	Complementary error function
Math.exp, Math.exp!	Base x of Euler
Math.frexp	Normalized fraction and exponent
Math.hypot	Hypotenuse
Math.ldexp	Floating-point value corresponding to mantissa and exponent
Math.sinh, Math.sinh!	Hyperbolic sine
Math.sqrt, Math.sqrt!	Square root
Math.tan, Math.tan!	Tangent
Math.tanh, Math.tanh!	Hyperbolic tangent

15.3 Some Examples

Now that we have a list of the math methods available to us, we can start to use them:

To perform a square root:

```
Math.sqrt(9)
=> 3.0
```

Or a Euler calculation:

```
Math.exp(2)
=> 7.38905609893065
```

15.4 Summary

This chapter has covered the concepts behind Ruby methods and functions. The next chapter will focus on Ruby logical operators.

Chapter 16.   Understanding Ruby Logical Operators

The objective of this chapter is to provide an overview of Ruby Logical Operators

16.1 Ruby Logical Operators

Logical Operators are also known as *Boolean Operators* because they evaluate parts of an expression and return a *true* or *false* value, allowing decisions to be made about how a program should proceed.

Rather than to look at a code example right away, the first step to understanding how logical operators work in Ruby is to construct a sentence. Let's assume we need to check some aspect of two variables named *var1* and *var2*. Our sentence might read:

> *If var1 is less than 25 AND var2 is greater than 45 then return true.*

Here the logical operator is the "AND" part of the sentence. If we were to express this in Ruby we would use the comparison operators we covered earlier together with the *and* or *&&* logical operators:

```
var1 = 20
var2 = 60
var1 < 25 and var2 > 45
=> true
```

Similarly, if our sentence was to read:

> *If var1 is less than 25 OR var2 is greater than 45 return true.*

Then we would replace the "OR" with the Ruby equivalent *or*, or '||':

```
var1 < 25 or var2 > 45
=> true
```

Another useful *Logical Operator* is the *not* operator which simply inverts the result of an expression. The ! character represents the NOT operator and can be used as follows:

```
10 == 10        # returns ''true''
=> true
```

```
!(10 == 10)        // returns ''false'' because we have inverted the result
with the not operator
=> false
```

Logical operators are of particular use when constructing conditional code, a topic which will be covered in detail in *Ruby Flow Control*.

Chapter 17.   Ruby Object Oriented Programming

Ruby is an object oriented environment and, as such, provides extensive support for developing object-oriented applications. The area of object oriented programming is, however, large. Entire books can, and indeed have, been dedicated to the subject. A detailed overview of object oriented software development is beyond the scope of Ruby Essentials. Instead we will introduce the basic concepts involved in object oriented programming and then move on to explaining the concept as it relates to Ruby development.

17.1 What is an Object?

An object is a self-contained piece of functionality that can be easily used, and re-used as the building blocks for a software application.

Objects consist of data variables and functions (called *methods*) that can be accessed and called on the object to perform tasks. These are collectively referred to as *members*.

Just about everything in Ruby, from numbers and strings to arrays is an object.

17.2 What is a Class?

Much as a blueprint or architect's drawing defines what an item or a building will look like once it has been constructed, a class defines what an object will look like when it is created. It defines, for example, what the *methods* will do and what the *member* variables will be.

New classes can be created based on existing classes, a concept known as *inheritance*. In such a scenario the new class (known as the *subclass*) inherits all the features of the parent class (known as the *superclass*) with added features that differentiate it from the *superclass*. Ruby supports *single inheritance* in that a subclass can only inherit from a single *superclass*.

Other languages such as C++ support *multiple inheritance* where the *subclass* can inherit the characteristics of multiple *superclasses*.

17.3 Defining a Ruby Class

For the purposes of this tutorial we will create a new class intended to be used as part of a banking application. Classes are defined using the *class* keyword followed by the *end* keyword and must be given a name by which they can be referenced. This name is a constant so must begin with a capital letter.

With these rules in mind we can begin work on our class definition:

```
class BankAccount
    def initialize ()
    end


    def test_method
        puts "The class is working"
    end
end
```

The above example defines a class named *BankAccount* with a single method called *test_method* which simply outputs a string. The *initialize* method is a standard Ruby class method and is the method which gets called first after an object based on this class has completed initialization. You can place any code in the initialize method and it is particularly useful for passing in arguments when the class is created.

17.4 Creating an Object from a Class

An object is created from a class using the *new* method. For example, to create an instance of our BankAccount class we simply perform the following:

```
account = BankAccount.new()
```

This creates a BankAccount object named *account*. Having created the object we can call our test_method:

```
account.test_method
The class is working
```

17.5 Instance Variables and Accessor Methods

Instance variables are variables defined in a class that are available only to each instance of the class. Instance variables may be defined either inside or outside of class methods. To make the variables available from outside the class, they must be defined within *accessor methods* (also known as a *getter* method).

For example, we might want to add instance variables to our *BankAccount* class:

```
class BankAccount
```

```
    def accountNumber
         @accountNumber = "12345"
    end


    def accountName
         @accountName = "John Smith"
    end


    def initialize ()
    end


    def test_method
         puts "The class is working"
         puts accountNumber
    end
end
```

Now we have two instance variables called *@accountNumber* and *@accountName* with associated accessor methods. We can now access these variables from outside the

```
account = BankAccount.new()
puts account.accountNumber
puts account.accountName
```

The two *puts* statements above will print out the values of the two variables returned by the accessor methods (in this case "12345" and "John Smith" respectively).

Now that we can "get" the value of an instance variable, we now need a way to "set" the value of an instance variable. One way to do this is via a *setter* methods. Let's clean up our BankAccount class, so that it has two instance variables with getters and setters:

```
class BankAccount


    def accountNumber
         @accountNumber
```

```
    end

    def accountNumber=( value )
        @accountNumber = value
    end

    def accountName
        @accountName
    end

    def accountName=( value )
        @accountName = value
    end

end
```

We can now create an instance of our class, set the name and account number using the setters and then access them using the getters:

```
account = BankAccount.new()

account.accountNumber = "54321"
account.accountName = "Fred Flintstone"

puts account.accountNumber
puts account.accountName
```

17.6 **Ruby Class Variables**

A *class variable* is a variable shared between all instances of a class. In other words, there is one instance of the variable and it is accessed by object instances. An instance variable must be initialized within the class definition. Class variables are prefixed with two @ characters (@@).

To demonstrate this we will add an *@@interest_rate* class variable (since the same interest rate applies to all back accounts):

```
class BankAccount
```

```
    def interest_rate
        @@interest_rate = 0.2
    end


    def accountNumber
        @accountNumber
    end


    def accountNumber=( value )
        @accountNumber = value
    end


    def accountName
        @accountName
    end


    def accountName=( value )
        @accountName = value
    end


end
```

17.7 Instance Methods

Although we looked briefly at instance methods earlier in this chapter we have so far focused on storage of data in a class.

Instance methods are methods that can be called on an instance of the class. Instance methods can access class variables to perform tasks, and can also accept values as arguments. For example, we can add a method to our class that takes a new account balance as an argument and use the @@interest_rate class variable to calculate the interest due:

```
    def calc_interest ( balance )
        puts balance * interest_rate
    end
```

Now, when we create an instance of our class, we can call the new method:

```
account = BankAccount.new()
account.calc_interest( 1000 )
700.0
```

17.8 Ruby Class Inheritance

As we mentioned earlier in this chapter, Ruby supports *single inheritance*. This means that a subclass can be created that inherits all the variables and methods of another class. The subclass is then extended to add new methods or variables not available in the superclass.

One class inherits from another using the < character. Say, for example, that we want a new kind of BankAccount class. This class needs all the same variables and methods as our original class, but also needs the customer's phone number. To do this we simply inherit from *BankAccount* and add the new instance variable:

```
class NewBankAccount < BankAccount

   def customerPhone
        @customerPhone
   end


   def customerPhone=( value )
        @customerPhone = value
   end

end
```

We now have a new class, derived from BankAccount. This new subclass has everything the superclass had, plus a new property - the customer's phone number:

```
account.accountNumber = "54321"
account.customerPhone = "555-123-5433"
54321
555-123-5433
```

Note that the above example assumes the declaration of the BankAccount class is in the same Ruby source file as the NewBankAccount declaration. If this is not the case, the *require* statement must be used to tell Ruby which file to include to find the BankAccount class. Assuming that BankAccount is defined in a file named "BankAccount.rb" we would include the file as follows:

```ruby
require 'BankAccount'

class NewBankAccount < BankAccount


    def customerPhone
         @customerPhone
    end


    def customerPhone=( value )
         @customerPhone = value
    end


end
```

Chapter 18.   Ruby Flow Control

One of the most powerful features of Ruby (and every other programming or scripting language for that matter) is the ability to build intelligence and logic into code. This is achieved through the use of *control structures* that decide what code is executed based on logical expressions.

In this chapter of Ruby Essentials we will look at how such control structures are built.

18.1 The Ruby if Statement

The *if* statement is the most basic of the Ruby control structures. *if* statements simply specify a section of Ruby code to be executed when a specified criteria is met. The syntax for an *if* statement is as follows:

```
if expression then
   ruby code
end
```

In the above outline, the *expression* is a logical expression that will evaluate to either true or false. If the expression is true, then the code represented by *ruby code* will execute. Otherwise the code is skipped. The *end* statement marks the end of the *if* statement.

Let's look at an example:

```
if 10 < 20 then
    print "10 is less than 20"
end
```

When executed, the code will display the string "10 is less than 20", because the 10 < 20 expression evaluated to true.

If this was any language other than Ruby we would now move on to the next section. Except that this is Ruby we are talking about, so we have more flexibility. Firstly, we can drop the *then* keyword and get the same result:

```
if 10 < 20
    print "10 is less than 20"
end
```

We can also place the *if* after the *print* statement, and in doing so, drop the *end* statement:

```
print "10 is less than 20" if 10 < 20
```

Similarly, we can place a colon (:) between the *if* expression and the statement to be executed:

```
if 10 < 20: print "10 is less than 20" end
```

The expression to be evaluated can also include logical operators. For example:

```
if firstname == "john" && lastname == "smith" then
print "Hello John!"
end
```

18.2 Using else and elsif Constructs

The *if* control structure outlined above allows you to specify what should happen if a particular expression evaluates to *true*. It does not, however, provide the option to specify something else that should happen in the event that the expression evaluates to *false*. This is where the *if ... else* construct comes into play.

The syntax for *if ... else* is similar to the simple *if* statement with the exception that the *else* statement can be used to specify alternate action:

```
if customerName == "Fred"
      print "Hello Fred!"
else
      print "You're not Fred! Where's Fred?"
end
```

As shown in the above example the code block following the *if* statement is executed when the expression is evaluated to be true (i.e. the customerName variable contains the string "Fred") and the script after the *else* statement is executed when the customerName does not match the string "Fred".

elsif structures take the *if ... else ...* concept a step further to implement *if ... else ... if ... else ...* structures. For example:

```
if customerName == "Fred"

      print "Hello Fred!"

elsif customerName == "John"

      print "Hello John!"

elsif customername == "Robert"

      print "Hello Bob!"

end
```

This construct may also be expressed a little more economically using a colon (:) to separate the *elsif* from the *print* statement:

```
if customerName == "Fred" : print "Hello Fred!"

elsif customerName == "John" : print "Hello John!"

elsif customername == "Robert" : print "Hello Bob!"

end
```

18.3 The Ruby Ternary Operator

Ruby uses something called a *ternary operator* to provide a shortcut way of making decisions. The syntax of the ternary operator is as follows:

```
[condition] ? [true expression] : [false expression]
```

The way this works is that *[condition]* is replaced with an expression that will return either true or false. If the result is true then the expression that replaces the *[true expression]* is evaluated. Conversely, if the result was false then the *[false expression]* is evaluated. Let's see this in action:

```
irb(main):182:0> customerName = "Fred"

=> "Fred"

irb(main):183:0> customerName == "Fred" ? "Hello Fred" : "Who are you?"

=> "Hello Fred"
```

18.4 Summary

In this chapter of Ruby Essentials we have explored basic control structures which allow us to build decision making logic into our Ruby programs. In the following chapter we will look at a more advanced conditional structure, *The Ruby case Statement*.

Chapter 19.   The Ruby case Statement

In this chapter of Ruby Essentials we will look at implementing flow control logic in Ruby scripts through the use of the Ruby *case* statement.

19.1 Ruby Flow Control

In the previous chapter we looked at some basic conditional structures using the *if ... else* and *if .. elsif ...* mechanisms. These approaches to building conditional logic work well if you need to check a value against only a few different criteria (for example checking the value of a string against a couple of possible candidates):

```
if customerName == "Fred"
      print "Hello Fred!"
elsif customerName == "John"
      print "Hello John!"
elsif customername == "Robert"
      print "Hello Bob!"
end
```

This can quickly become cumbersome, however, when a need arises to evaluate a large number of conditions. A much easier way to handle such situations is to use the Ruby *case* statement, the syntax for which is defined as follows:

```
result = case value
    when match1 then result1
    when match2 then result2
    when match3 then result3
    when match4 then result4
    when match5 then result5
    when match6 then result6
    else result7
end
```

There can be any number of *when* statements - basically as many as you need to fully compare the *value* in the *case* statement against the possible options (represented by *match1* through to *match7* in the above example) specified by the *when* statements. When a match is found the result is assigned to the optional *result* variable.

Finally, the *else* statement specifies the default result to be returned if no match is found.

This concept is, perhaps, best explained using an example. The following Ruby case statement is designed to match a particular car model with a manufacturer. Once a match is found, the car and associated manufacturer are included in an output string:

```
car = "Patriot"

manufacturer = case car
    when "Focus" then "Ford"
    when "Navigator" then "Lincoln"
    when "Camry" then "Toyota"
    when "Civic" then "Honda"
    when "Patriot" then "Jeep"
    when "Jetta" then "VW"
    when "Ceyene" then "Porsche"
    when "Outback" then "Subaru"
    when "520i" then "BMW"
    when "Tundra" then "Nissan"
    else "Unknown"
end

puts "The " + car  + " is made by "  + manufacturer
```

When executed, the resulting output will read:

```
The Patriot is made by Jeep
```

If no match was found in the case statement, then the default result, defined by the *else* statement would cause the following output to be generated:

```
The Prius is made by Unknown
```

19.2 Number Ranges and the case statement

The case statement is also particularly useful when used in conjunction with number ranges (for details of ranges read *Ruby Ranges*).

The following *case* example detects where a number falls amongst a group of different ranges:

```ruby
score = 70

result = case score
   when 0..40 then "Fail"
   when 41..60 then "Pass"
   when 61..70 then "Pass with Merit"
   when 71..100 then "Pass with Distinction"
   else "Invalid Score"
end

puts result
```

The above code, when executed will result in a "Pass with Merit" message.

19.3 Summary

The *if .. else ...* approach to building conditional logic into an application works fine for evaluating a limited number of possible criteria. For much larger evaluations, the Ruby *case* statement is a less cumbersome alternative. In the chapter we have looked at the case statement and reviewed some examples using strings and number ranges.

Chapter 20.   Ruby While and Until Loops

If computers do one thing well it is performing tasks very quickly. One thing computers do even better is performing the same task over and over again until specific criteria are met (or even infinitely if that is what is required). One thing that computers are very bad at, however, is doing anything without being told to do it. Given these facts, it should come as no surprise that just about every programming language, Ruby included, provide mechanisms for instructing a computer system to repeat tasks.

In this chapter of Ruby Essentials we will look at using *while* and *until* structures to allow loops to be built into applications. In the next chapter we will look at using the *for loop* and Ruby looping methods.

20.1 The Ruby While Loop

The Ruby *while* loop is designed to repeat a task until a particular expression is evaluated to be *false*. The syntax of a *while* loop is as follows:

```
while expression do

... ruby code here ...

end
```

In the above outline, *expression* is a Ruby expression which must evaluate to *true* or *false*. The *ruby code here* marker is where the code to executed is placed. This code will be repeatedly executed until the *expression* evaluates to *false*.

For example, we might want to loop until a variable reaches a particular value:

```
i = 0
while i < 5 do
    puts i
    i += 1
end
```

The above code will output the value of *i* until *i* is no longer less than 5, resulting in the following output:

```
0

1

2

3

4
```

The *do* in this case is actually optional. The following is perfectly valid:

```
i = 0

while i < 5

    puts i

    i += 1

end
```

20.2 Breaking from While Loops

It is sometimes necessary to break out of a while loop before the *while* expression evaluates to false. This can be achieved using the *break if* statement:

```
i = 0

while i < 5

    puts i

    i += 1

    break if i == 2

end
```

The above loop will now exit when *i* equals 2, instead of when *i* reaches 5.

20.3 unless and until

Ruby's *until* statement differs from *while* in that it loops until a *true* condition is met. For example:

```
i = 0

until i == 5

    puts i

    i += 1
```

```
end
```

When executed, the above example produces the following output:

```
0
1
2
3
4
```

The *until* keyword can also be used as a statement modifier, as follows:

```
puts i += 1 until i == 5
```

The *unless* statement provides an alternative mechanism to the *if else* construct. For example we might write an *if else* statement as follows:

```
if i < 10
    puts "Student failed"
else
    puts "Student passed"
end
```

The *unless* construct inverts this:

```
unless i >= 10
    puts "Student failed"
else
    puts "Student passed"
end
```

20.4 Summary

In this chapter of Ruby Essentials we have looked in detail at creating loops that depend on specific criteria. This essentially involved the use of *while* and *until* statements. In the next chapter we will look at looping with *for* and the Ruby Looping Methods.

Chapter 21.   Looping with for and the Ruby Looping Methods

In the previous chapter we looked at Ruby *While* and *Until* loops as a way to repeat a task until a particular expression evaluated to *true* or *false*. In this chapter we will look at some other mechanisms for looping in a Ruby program, specifically *for loops* and a number of built-in methods designed for looping, specifically the *loop*, *upto*, *downto* and *times* methods.

21.1 The Ruby for Loop

The *for* loop is a classic looping construct that exists in numerous other programming and scripting languages. It allows a task to be repeated a specific number of times. Ruby differs in that it is used in conjunction with ranges (see *Ruby Ranges* for more details). For example, we can repeat a task 8 times using the following *for* statement:

```
for i in 1..8 do
    puts i
end
```

The above loop will result in the following output:

```
1
2
3
4
5
6
7
8
```

The *do* in the *for* statement is optional, unless the code is placed on a single line:

```
for i in 1..8 do puts i end
```

Ruby *for* loops can be nested:

```
for j in 1..5 do
    for i in 1..5 do
```

```
        print i,  " "
     end
puts
end
```

The above code will result in the following output:

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Also, the *break if* statement can be used to break out of a *for loop* (note that only the inner for loop is exited, if the loop is nested the outer loop will continue the looping run):

```
for j in 1..5 do
     for i in 1..5 do
         print i,  " "
         break if i == 2
     end
end
```

This causes the inner loop to break each time *i* equals 2, thereby returning control to the outer loop which in turn calls the inner loop:

```
1 2
1 2
1 2
1 2
1 2
```

21.2 The Ruby times Method

The *times* method provides an extremely convenient alternative to the *for* loop. The *times* method is provided by the Integer class and allows a task to be performed a specific number of times:

```
5.times { |i| puts i }


0

1

2

3

4
```

The above statement is identical to the following for loop, although moderately easier to type:

```
for i in 0..4
    puts i
end


0

1

2

3

4
```

21.3 The Ruby upto Method

The *upto* method can be called on Integer, String and Date classes and provides similar functionality to the *for* loop described earlier in this chapter.

For example, we might express a *for* loop as follows:

```
for i in 1..5 do
   puts i
end
```

Alternatively, we could achieve identical behavior using the *upto* method, passing the value to which we want the loop to run as an argument to the method:

```
5.upto(10) do
   puts "hello"
end
```

If we so desired, we could shorten this to a single line of code:

```
1.upto(5) { |i| puts i }
```

21.4 The Ruby downto Method

The *downto* method is similar to the *upto* method with the exception that it starts at a value and counts down, rather than up. For example, we might want a loop to execute for numbers between 15 and 10:

```
15.downto(10) {|i| puts i }
15
14
13
12
11
10
```

Chapter 22.   Ruby Strings - Creation and Basics

A string is a group of characters that typically make up human readable words or sentences. Because strings are essentially the mechanism by which applications communicate with their users it is inevitable that string manipulation is a key part of programming. In this chapter of Ruby Essentials we will cover the basics of working with strings in Ruby.

22.1 Creating Strings in Ruby

Strings are stored in Ruby using the *String* object. In addition to providing storage for strings, this object also contains a number of methods which can be used to manipulate strings.

A new string can be created using the *new* method of the *String* class:

```
myString = String.new
=> ""
```

The above method creates an empty string. Alternatively a new string may be created and initialized by passing through a string as an argument to the *new* method:

```
myString = String.new("This is my string. Get your own string")
```

Another option is to use the *String* method provided by the *Kernel*:

```
myString = String("This is also my string")
```

But, perhaps the easiest way to create a string is to simply assign a string to a variable name. Ruby then takes care of the rest for you:

```
myString = "This is also my string"
```

As far as creating string objects goes that is as easy as it gets!

22.2 Quoting Ruby Strings

Strings can be delimited using either double quotes (") or single quotes ('). Whilst both serve the purpose of encapsulating a string they have different purposes. The double quotes are designed to interpret escaped characters such as new lines and tabs so that they appear as

actual new lines and tabs when the string is rendered for the user. Single quotes, however, display the actual escape sequence, for example displaying \n instead of a new line.

The following example demonstrates the double quote in action:

```
myString = "This is also my string.\nGet your own string"


puts myString
This is also my string.
Get your own string
```

As you can see, the \n was interpreted as a real new line causing our string to appear on two lines when printed.

The single quote gives us a different result:

```
myString = 'This is also my string.\nGet your own string'


puts myString
This is also my string.\nGet your own string
```

In this case the '\n' is treated literally as a '\' and an 'n' with no special meaning.

22.3 General Delimited Strings

Ruby allows you to define any character you want as a string delimiter simply by prefixing the desired character with a %. For example, we could use the ampersand to delimit our string:

```
myString = %&This is my String&
```

This enables us to avoid quotes and double quotes embedded in a string being interpreted as delimiters:

```
myString = %&This is "my" String&
puts myString
This is "my" String
```

It is also possible to define delimiter pairs such as parentheses, braces or square brackets:

```
myString = %(This is my String)
myString = %[This is my String]
myString = %{This is my String}
```

Ruby also provides a few special delimited strings. %Q is the equivalent of double quote delimiters and %q is the equivalent of single quotes. %x is the equivalent of back-quote (`) delimited strings.

An easy way to embed quotes in a Ruby string is to *escape* them by preceding them with a backslash (\\):

```
myString = "This is \"my\" String"

myString = 'This is \'my\' String'
```

Another option, if you aren't using escape characters such as new lines (\n) is to use single quotes to delimit a string containing double quotes and vice versa:

```
myString = 'This is "my" String'

myString = "This is 'my' String"
```

22.4 Ruby Here Documents

A *Here Document* (or *heredoc* as it is more commonly referred to) provides a mechanism for creating free format strings, preserving special characters such as new lines and tabs.

A here document is created by preceding the text with << followed by the delimiter string you wish to use to mark the end of the text. The following example uses the string "DOC" as the delimiter:

```
myText = <<DOC
Please Detach and return this coupon with your payment.
Do not send cash or coins.

Please write your name and account number on the check and
make checks payable to:
```

```
        Acme Corporation


Thank you for your business.
DOC
```

When this string is printed it appears exactly as it was entered, together with all the new lines and tabs:

```
puts myText


Please Detach and return this coupon with your payment.
Do not send cash or coins.


Please write your name and account number on the check and
make checks payable to:


        Acme Corporation


Thank you for your business.
```

22.5 Getting Information about String Objects

The String object includes a number of methods that can be used to obtain information about the string. For example, we can find if a string is empty using the *empty?* method:

```
myString = ""
=> ""


myString.empty?
=> true
```

It is also possible to find the length of a string using the *length* and *size* methods:

```
myString = "Hello"
```

```
myString.length
=> 5
```

Chapter 23.   Ruby String Concatenation and Comparison

In the previous chapter (*Ruby Strings - Creation and Basics*) we looked at how to create a Ruby string object. In this chapter we will look at accessing, comparing and concatenating strings in Ruby. In the next chapter we will look at manipulating and converting Ruby strings.

23.1 Concatenating Strings in Ruby

If you've read any of the preceding chapters in this book you will have noticed that Ruby typically provides a number of different ways to achieve the same thing. Concatenating strings is certainly no exception to this rule.

Strings can be concatenated using the + method:

```
myString = "Welcome " + "to " + "Ruby!"
=> "Welcome to Ruby!"
```

In the interests of brevity, you can even omit the + signs:

```
myString = "Welcome " "to " "Ruby!"
=> "Welcome to Ruby!"
```

If you aren't happy with the above options you can chain strings together using the << method:

```
myString = "Welcome " << "to " << "Ruby!"
=> "Welcome to Ruby!"
```

Still not enough choices for you? Well, how about using the *concat* method:

```
myString = "Welcome ".concat("to ").concat("Ruby!")
=> "Welcome to Ruby!"
```

23.2 Freezing a Ruby String

A string can be *frozen* after it has been created such that it cannot subsequently be altered. This is achieved using the *freeze* method of the *String* class:

```
myString = "Welcome " << "to " << "Ruby!"
```

```
=> "Welcome to Ruby!"

myString.freeze

myString << "hello"
TypeError: can't modify frozen string
```

23.3 Accessing String Elements

Fragments of a Ruby string can be accessed using the [] method of the *String class*. One use for this method is to find if a particular sequence of characters exists in a string. If a match is found the sequence is returned, otherwise *nil* is returned:

```
myString = "Welcome to Ruby!"

myString["Ruby"]
=> "Ruby"

myString["Perl"]
=> nil
```

Pass an integer through to the [] method and the ASCII code of the character at that location in the string (starting at zero) will be returned. This can be converted back to a character using the *chr* method:

```
myString[3].chr
=> "c"
```

You can also pass through a start position and a number of characters to extract a subsection of a string:

```
myString[11, 4]
=> "Ruby"
```

You can also use a Range to specify a group of characters between start and end points:

```
myString[0..6]
=> "Welcome"
```

The location of a matching substring can be obtained using the *index* method:

```
myString.index("Ruby")
=> 11
```

23.4 Comparing Ruby Strings

It is not uncommon to need to compare two strings, either to assess equality or to find out if one string is higher or lower than the other (alphabetically speaking).

Equality is performed either using the == or *eql?* methods:

```
"John" == "Fred"
=> false


"John".eql? "John"
=> true
```

The spaceship (<=>) method can be used to compare two strings in relation to their alphabetical ranking. The <=> method returns 0 if the strings are identical, -1 if the left hand string is less than the right hand string, and 1 if it is greater:

```
"Apples" <=> "Apples"
=> 0


"Apples" <=> "Pears"
=> -1


"Pears" <=> "Apples"
=> 1
```

23.5 Case Insensitive String Comparisons

A case insensitive comparison may be performed using the *casecmp* method which returns the same values as the <=> method described above:

```
"Apples".casecmp "apples"
=> 0
```

Chapter 24.   Ruby String Replacement, Substitution and Insertion

In this chapter we will look at some string manipulation techniques available in Ruby. Such techniques include string substitution, multiplication and insertion. We will also take a look at the Ruby *chomp* and *chop* methods.

24.1 Changing a Section of a String

Ruby allows part of a string to be modified through the use of the []= method. To use this method, simply pass through the string of characters to be replaced to the method and assign the new string. As is often the case, this is best explained through the use of an example:

```
myString = "Welcome to JavaScript!"


myString["JavaScript"]= "Ruby"


puts myString
=> "Welcome to Ruby!"
```

As you can see, we replaced the word "JavaScript" with "Ruby".

The []= method can also be passed an index representing the index at with the replacement is to take place. In this instance the single character at the specified location is replaced by the designated string:

```
myString = "Welcome to JavaScript!"
myString[10]= "Ruby"


puts myString
=> "Welcome toRubyJavaScript!"
```

Perhaps a more useful trick is to specify an index range to be replaced. For example we can replace the characters from index 8 through to index 20 inclusive:

```
myString = "Welcome to JavaScript!"
=> "Welcome to JavaScript!"
```

```
myString[8..20]= "Ruby"

=> "Ruby"


puts myString

=> "Welcome Ruby!"
```

24.2 Ruby String Substitution

The *gsub* and *gsub!* methods provide another quick and easy way of replacing a substring with another string. These methods take two arguments, the search string and the replacement string. The *gsub* method returns a modified string, leaving the original string unchanged, whereas the *gsub!* method directly modify the string object on which the method was called:

```
myString = "Welcome to PHP Essentials!"

=> "Welcome to PHP Essentials!"


myString.gsub("PHP", "Ruby")

=> "Welcome to Ruby Essentials!"
```

An entire string, as opposed to a substring, may be replaced using the *replace* method:

```
myString = "Welcome to PHP!"

=> "Welcome to PHP!"


myString.replace "Goodbye to PHP!"

=> "Goodbye to PHP!"
```

24.3 Repeating Ruby Strings

A string may be multiplied using the * method (not something I have ever needed to do myself, but the capability is there if you need it):

```
myString = "Is that an echo? "

=> "Is that an echo? "


myString * 3
```

```
=> "Is that an echo? Is that an echo? Is that an echo? "
```

24.4 Inserting Text into a Ruby String

So far in this chapter we have looked exclusively at changing the existing text contained in a Ruby string object. Another common requirement is to insert new text at a certain location in a string. This is achieved in Ruby using the *insert* method. The insert method takes as arguments index position into the string where the insertion is take place, followed by the string to be inserted

```
myString = "Paris in Spring"

myString.insert 8, " the"
=> "Paris in the Spring"
```

24.5 Ruby chomp and chop Methods

The purpose of the *chop* method is to remove the trailing character from a string:

```
myString = "Paris in the Spring!"
=> "Paris in the Spring!"

myString.chop
=> "Paris in the Spring"
```

Note that *chop* returns a modified string, leaving the original string object unchanged. Use *chop!* to have the change applied to the string object on which the method was called.

The *chomp* method removes *record separators* from a string. The record separator is defined by the $/ variable and is, by default, the new line character (\n). As with the *chop* method the *chomp!* variant of the method applies the change to string object on which the method is called:

```
myString = "Please keep\n off the\n grass"
=> "Please keep\n off the\n grass\n"

myString.chomp!
```

```
=> "Please keep\n off the\n grass"
```

24.6 Reversing the Characters in a String

The *reverse* method is used to reverse the contents of a string:

```
myString = "Paris in the Spring"
=> "Paris in the Spring"


myString.reverse
=> "gnirpS eht ni siraP"
```

Once again, not something I've ever needed to do in all my years as a developer, but you never know when you might need to do it.

Chapter 25.   Ruby String Conversions

So far we have looked at creating, comparing and manipulating strings. Now it is time to look at converting strings in Ruby.

25.1 Converting a Ruby String to an Array

Strings can be converted to arrays using a combination of the *split* method and some regular expressions. The *split* method serves to break up the string into distinct parts that can be placed into array element. The regular expression tells *split* what to use as the break point during the conversion process.

We can start by converting an entire string into an array with just one element (i.e. the entire string):

```
myArray = "ABCDEFGHIJKLMNOP".split
=> ["ABCDEFGHIJKLMNOP"]
```

This has created an array object called *myArray*. Unfortunately this isn't much use to us because we wanted each character in our string to be placed in an individual array element. To do this we need to specify a regular expression. In this case we need to use the regular expression which represents the point between two characters (//) and pass it through as an argument to the *split* method:

```
myArray = "ABCDEFGHIJKLMNOP".split(//)
=> ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N",
"O", "P"]
```

We can also create an array based on words by using the space (/ /) as the break point:

```
myArray = "Paris in the Spring".split(/ /)
=> ["Paris", "in", "the", "Spring"]
```

Or we can even, perhaps most usefully, convert a comma separated string into an array:

```
myArray = "Red, Green, Blue, Indigo, Violet".split(/, /)
=> ["Red", "Green", "Blue", "Indigo", "Violet"]
```

25.2 Changing the Case of a Ruby String

The first letter of a string (and only the first letter of a string) can be capitalized using the *capitalize* and *capitalize!* methods (the first returns a modified string, the second changes the original string):

```
"paris in the spring".capitalize
=> "Paris in the spring"
```

The first character of each word in a Ruby string may be capitalized by iterating through the string (assuming the record separator is a new line):

```
"one\ntwo\nthree".each {|word| puts word.capitalize}
One
Two
Three
=> "one\ntwo\nthree"
```

The case of every character in a string may be converted using the *upcase*, *downcase* and *swapcase* methods. These methods are somewhat self explanatory, but here are some examples for the avoidance of doubt:

```
"PLEASE DON'T SHOUT!".downcase
=> "please don't shout!"


"speak up. i can't here you!".upcase
=> "SPEAK UP. I CAN'T HERE YOU!"


"What the Capital And Lower Case Letters Swap".swapcase
=> "wHAT THE cAPITAL aND lOWER cASE lETTERS sWAP"
```

25.3 Performing String Conversions

Strings can be converted to other Ruby object types such as *Fixnums*, *Floats* and *Symbols*.

To convert a String to an integer use the *to_i* method:

```
"1000".to_i
```

```
=> 1000
```

To convert a String to Float use the *to_f* method:

```
"1000".to_f
=> 1000.0
```

To convert a String to a Symbol use the *to_sym* method:

```
"myString".to_sym
=> :myString
```

25.4 **Summary**

This chapter has covered in detail the process of converting strings in Ruby. The next chapter will look at working with file system directories with Ruby.

Chapter 26.   Ruby Directory Handling

It may have escaped your notice, but up until this chapter everything we have done involved working with data in memory. Now that we have covered all the basics of the Ruby language, it is time to turn our attention to working with files and directories in Ruby.

26.1 Changing Directory in Ruby

When a Ruby application is launched it is usually done from a particular directory. Often it is necessary to navigate to a different directory somewhere on the file system from within the Ruby code. Ruby provides a number of useful directory navigation methods in the *Dir* class.

First, it is often useful to identify the current directory. This can be done with the *pwd* method of the Ruby *Dir* class:

```
Dir.pwd => "/home/ruby"
```

Changing the current working directory in Ruby is achieved using the *chdir* method of the Ruby *Dir* class. This method takes the path of the destination directory as an argument:

```
Dir.chdir("/home/ruby/test")
```

26.2 Creating New Directories

Directory creation in Ruby is handled by the *mkdir* method of the *Dir* class. This method takes as its argument the path of the new directory. This can either be a full path to the directory, or a relative path based on the current working directory:

```
Dir.mkdir("/home/ruby/temp")
=> 0
```

26.3 Directory Listings in Ruby

Once we have navigated to the desired directory it is a common requirement to obtain a listing of files contained within that directory. Such a listing can be obtained using the *entries* method. The *entries* method takes as an argument the path of the directory for which a listing is required and returns an array containing the names of the files in that directory:

In the following example we request a listing of the files in the current directory, which is represented by a dot (.).

```
Dir.entries(".")
```

```
=> ["techotopia_stats.jpg", "toolButton.png", ".", "..",
"techotopia_stats_since_start.jpg", "music_728x90_1.jpg",
"music_468x60_a.jpg", "Fedora_essentials.jpg"]
```

We can use some of the techniques covered in *Understanding Ruby Arrays* to extract the elements from the array:

```
dirListing.each { |file| puts file }
techotopia_stats.jpg
toolButton.png
.
..
techotopia_stats_since_start.jpg
music_728x90_1.jpg
music_468x60_a.jpg
Fedora_essentials.jpg
```

Alternatively, we can utilize the *foreach* method of the *Dir* class to achieve the same result:

```
Dir.foreach(".") { |file| puts file }
techotopia_stats.jpg
toolButton_IST.png
.
..
techotopia_stats_since_start.jpg
music_728x90_1.jpg
music_468x60_a.jpg
Fedora_essentials.jpg
```

26.4 Summary

This chapter has covered the basics of directory handling in Ruby. The next chapter will cover the concepts of Ruby file handling.

Chapter 27.   Working with Files in Ruby

In the previous chapter we looked at how to work with directories. This chapter will look in detail at how to create, open and read and write to files in Ruby. We will then learn how to delete and rename files.

27.1 Creating a New File with Ruby

New files are created in Ruby using the *new* method of the *File* class. The *new* method accepts two arguments, the first being the name of the file to be created and the second being the mode in which the file is to opened. Supported file modes are shown the following table:

Mode	Description
r	Read only access. Pointer is positioned at start of file.
r+	Read and write access. Pointer is positioned at start of file.
w	Write only access. Pointer is positioned at start of file.
w+	Read and write access. Pointer is positioned at start of file.
a	Write only access. Pointer is positioned at end of file.
a+	Read and write access. Pointer is positioned at end of file.
b	Binary File Mode. Used in conjunction with the above modes. Windows/DOS only.

With this information in mind we can, therefore, create a new file in "write" mode as follows:

```
File.new("temp.txt", "w")
=> #<File:temp.txt>
```

27.2 Opening Existing Files

Existing files may be opened using the *open* method of the *File* class:

```
file = File.open("temp.txt")
=> #<File:temp.txt>
```

Note that existing files may be opened in different modes as outlined in the table above. For example, we can open a file in read-only mode:

```
file = File.open("temp.txt", "r")
=> #<File:temp.txt>
```

It is also possible to identify whether a file is already open using the *closed?* method:

```
file.closed?
=> false
```

Finally, we can close a file using the *close* method of the Ruby *File* class:

```
file = File.open("temp.txt", "r")
=> #<File:temp.txt>
file.close
=> nil
```

27.3 Renaming and Deleting Files in Ruby

Files are renamed and deleted in Ruby using the *rename* and *delete* methods respectively. For example, we can create a new file, rename it and then delete it:

```
File.new("tempfile.txt", "w")
=> #<File:tempfile.txt>

File.rename("tempfile.txt", "newfile.txt")
=> 0

File.delete("newfile.txt")
=> 1
```

27.4 Getting Information about Files

File handling often requires more than just opening files. Sometimes it is necessary to find out information about a file before it is opened. Fortunately the *File* class provides a range of methods for this very purpose.

To find out if a file already exists, use the *exists?* method:

```
File.exists?("temp.txt")
=> true
```

To find out if the file is really a file as opposed to, for example, a directory use the *file?* method:

```
File.file?("ruby")
=> false
```

Similarly, find out if it is a directory with the *directory?* method:

```
File.directory?("ruby")
=> true
```

To identify whether a file is readable, writable or executable, use the *readable?*, *writable?* and *executable?* methods:

```
File.readable?("temp.txt")
=> true


File.writable?("temp.txt")
=> true


File.executable?("temp.txt")
=> false
```

Get the size of a file with, yes you guessed it, the *size* method:

```
File.size("temp.txt")
=> 99
```

And find if a file is empty (i.e. zero length) with the *zero?* method:

```
File.zero?("temp.txt")
=> false
```

Find out the type of the file with the *ftype* method:

```
File.ftype("temp.txt")
=> "file"
```

```
File.ftype("../ruby")
=> "directory"


File.ftype("/dev/sda5")
=> "blockSpecial"
```

Finally, find out the create, modify and access times with *ctime*, *mtime* and *atime*:

```
File.ctime("temp.txt")
=> Thu Nov 29 10:51:18 EST 2007


File.mtime("temp.txt")
=> Thu Nov 29 11:14:18 EST 2007


File.atime("temp.txt")
=> Thu Nov 29 11:14:19 EST 2007
```

27.5 Reading and Writing Files

Once we've opened an existing file or created a new file we need to be able to read from and write to that file. We can read lines from a file using either the *readline* or *each* methods:

```
myfile = File.open("temp.txt")
=> #<File:temp.txt>


myfile.readline
=> "This is a test file\n"


myfile.readline
=> "It contains some example lines\n"
```

Alternatively, we can use the *each* method to read the entire file:

```
myfile = File.open("temp.txt")
=> #<File:temp.txt>

```

```
myfile.each {|line| print line }
This is a test file
It contains some example lines
But other  than that
It serves no real purpose
```

It is also possible to extract data from a file on a character by character basis using the *getc* method:

```
myfile = File.open("Hello.txt")
=> #<File:temp.txt>

myfile.getc.chr
=> "H"
myfile.getc.chr
=> "e"
myfile.getc.chr
=> "l"
```

Needless to say, we can also write to a file using the *putc* method to write a character at a time and *puts* to write a string at a time - note the importance of the *rewind* method call. This moves the file pointer back to the start of the file so we can read what have written:

```
myfile = File.new("write.txt", "w+")    # open file for read and write
=> #<File:write.txt>

myfile.puts("This test line 1")         # write a line
=> nil

myfile.puts("This test line 2")         # write a second line
=> nil

myfile.rewind                           # move pointer back to start of file
=> 0

myfile.readline
```

```
=> "This test line 1\n"


myfile.readline
=> "This test line 2\n"
```

Chapter 28.   Working with Dates and Times in Ruby

Ruby provides a *date* library containing the *Date* and *DateTime* classes, designed to provide mechanisms for manipulating dates and times in Ruby programs. The purpose of this chapter is to provide an overview of these classes.

28.1 Accessing the Date and DateTime Classes in Ruby

The *Date* and *DateClass* classes reside in the Ruby *date* library. Before these classes and their respective methods can be used, the *date* library must be included using the *require* directive*:*

```
require 'date'
```

28.2 Working with Dates in Ruby

Dates are handled in Ruby using the *Date* class. An instantiated Date object contains day, year and month values. A Date object may be initialized with day, month and year values on creation:

```
require 'date'

date = Date.new(2008, 12, 22)

date = Date.new(2008, 12, 22)
=> #<Date: 4909645/2,0,2299161>
```

Having created the date object we can access the properties of the object:

```
date.day
=> 22

date.month
=> 12

date.year
=> 2008
```

28.3 Working with Dates and Times

If both a date and a time value are needed, the *DateTime* class comes into use. As with the *Date* class, this class can similarly be pre-initialized on creation:

```
require 'date'
date = DateTime.new(2008, 12, 22, 14, 30)
```

Alternatively, the object can be configured to contain today's date and the current time:

```
date = DateTime.now
```

28.4 Calculating the Difference Between Dates

We can also calculate the difference between two dates down to the hour, minute and second:

```
require 'date'

today = DateTime.now
=> #<DateTime: 441799066630193/180000000,-301/1440,2299161>

birthday = Date.new(2008, 4, 10)
=> #<Date: 4909133/2,0,2299161>

days_to_go = birthday - today
time_until = birthday - today
=> Rational(22903369807, 180000000)

time_until.to_i           # get the number of days until my birthday
=> 127

hours,minutes,seconds,frac = Date.day_fraction_to_time(time_until)
[3053, 46, 57, Rational(1057, 180000000)]

puts "It is my birthday in #{hours} hours, #{minutes} minutes and #{seconds}
seconds (not that I am counting)"
```

```
It is my birthday in 3053 hours, 46 minutes and 57 seconds (not that I am
counting)
```