Free Study Notes on Ruby
By Satish Talim

Free Study Notes on Ruby
Copyright © 2006 - 2007
All rights reserved.

Author: **Satish Talim**

You may freely copy and distribute this eBook as long as you do not modify the text or remove this copyright notice. You must not make any charge for this eBook.

Notice: These notes are to be used for reference purposes only. The author will not accept any liability for any damage caused by the use of this material.

Revised Edition – 24th June 2007
First Edition – June 2006

About

Satish Talim is a senior software consultant based in Pune, India with over 29+ years of I.T. experience. His experience lies in developing and executing business for high technology and manufacturing industry customers. Personally his strengths lie in Business Development and Business Networking apart from new product and solution ideas. Good experience of organization development. Excellent cross disciplinary background in engineering, computer science and management.

He has -

- helped start subsidiaries for many US based software companies like Infonox (based in San Jose, CA), Maybole Technologies Pvt. Ltd. (Servient Inc. based in Houston, Texas) in Pune, India.
- been associated with Java / J2EE since 1995 and currently involved with Ruby and Rails.
- also started and manages two very active Java and Ruby User Groups in Pune, India - PuneJava and PuneRuby
- A Ruby Mentor on rubyforge.org, helping people get started with Ruby.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Table of Contents

Learn Ruby with me 9
Preamble 9
Assumptions 10
Using this eBook 10
What is Ruby? 13
How Ruby can help you, in more detail 14
Downloading Ruby and an Editor 14
Ruby Programming Environment 15
Our First Ruby program 18
Some Features of Ruby 23
Numbers in Ruby 25
Operators and Precedence 28
Fun with Strings 30
Variables and Assignment 32
Scope 36
Global scope and global variables 36
Built-in global variables 36

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Local scope 37
Getting input 38
Names in Ruby 40
More on methods 44
Writing Own Methods in Ruby 45
Bang (!) methods 51
Ruby Method Missing 54
More on the String class 56
Listing all methods of a class or object 58
Comparing two strings for equality 59
Simple Constructs in Ruby 62
Case Expressions 65
Ruby nil is an Object 66
Arrays in Ruby 67
Ranges 73
Blocks and Procs 75
Random Numbers 82
Reading from / Writing to text files 85
Traversing Directory Trees 86
Random Access 87

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Writing our own Class 90
Literal Constructors 96
Garbage Collection 97
Class Methods 98
Including Other Files 101
Open classes 105
Inheritance 108
Duck Typing 114
Overloading Methods 119
Overriding Methods 121
Usage of super 122
Redefining methods 124
Symbols 126
Hashes 130
Exploring Time class 133
Exceptions 135
Raising an Exception 135
Handling an Exception 139
Access Control 145
Top-level methods 149

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Accessor methods 151
Syntactic sugar 155
Mutable and Immutable Objects 160
Freezing Objects 160
Object Serialization 163
Constants 165
Modules/Mixins 170
Self - The current/default object 176
Regular Expressions 183
Literal characters 185
The wildcard character . (dot) 186
Character classes 186
Special escape sequences for common character classes 188
Unit Testing 193
Usage of TCPServer and TCPSocket Classes for Date and Time 209
Basic Networking 209
Port 210
Internet Addresses 211
Sockets 211
Socket classes 212

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The Date Time Server and Client 213
Summary 216
A Small Project using SMTP class 217
Web Services and Distributed Programming 220
Writing a SOAP Client 221
Writing a PuneRuby SOAP server and client 224
Ruby/Tk 227
Simple Tk applications 228
Using Ruby/MySQL 235
Ruby Tools 238
irb 238
The debugger 238
Profiling 239
ri and RDoc 240
ERb 241
Java and Ruby 243
Similarities 243
Differences 243
JRuby Tutorial 246
What is JRuby? 246

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Download and Setup 247
Where to use JRuby? 248
Resources: 250
Useful Articles 250
Ruby Quirks 251
Appendix A 260
Ruby Resources 260
Books/eBooks/Magazines 260
Useful Links 261
Blogs 263
Forums/User Groups 264
Companies in India working in Ruby/Rails 264
Appendix B 266
Solutions to Assignments 266
Appendix C 275

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Learn Ruby with me

Preamble

My interest in Ruby was aroused after I read an article Ruby the Rival in November 2005. I decided to learn Ruby myself and started making my study notes. What's presented here is a result of that.

I have made extensive references to the blogs and articles of various Ruby Gurus and to the Ruby books from Manning, O'Reilly and Pragmatic Programmers; my thanks to all of them.

I'd like to thank everyone on the ruby-talk mailing list for their thoughts and encouragement; all of my wonderful PuneRuby RUG members for their help in making these study notes far better than I could have done alone.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Assumptions

1. I am assuming that you know some programming language.
2. Ruby covered here will give you the grounding you need to understand Rails code.
3. The program examples are more tuned towards application- rather than systems-programming.

Using this eBook

This eBook is a step-by-step tutorial to programming in Ruby and you can follow it chapter by chapter, reading the text and running the sample programs. On the other hand, if you prefer to try out some of the programs in whichever order that takes your fancy; then refer back to the text for explanations. There are no large applications in this book – just small, self-contained sample programs.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Any source code in this eBook, is written like this:

```
def hello
  puts 'hello'
end
```

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

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

The blue box is for some notes related to **Rails**.

When there is a sample program to accompany the code, the program name is shown in a little box like this:

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

hello.rb

You can download the source code of all the programs shown in the eBook from here.

If you notice any errors or typos, or have any comments or suggestions or good exercises I could include, please email me.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

What is Ruby?

Ruby is a cross-platform interpreted and an object-oriented language. Ruby has been designed on the **Principle of Least Surprise** - Matz says "*I wanted to minimize my frustration during programming, so I want to minimize my effort in programming. That was my primary goal in designing Ruby. I want to have fun in programming myself. After releasing Ruby and many people around the world got to know Ruby, they said they feel the way I feel. They came up with the phrase the principle of least surprise*."

The year 2004 saw a massive surge of interest in Ruby, with the introduction of the Ruby on Rails Web application framework by David Heinemeier Hansson.

Yukihiro Matsumoto, commonly known as 'Matz' created the Ruby language in 1993.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

How Ruby can help you, in more detail

In David Black's book 'Ruby for Rails', he mentions that a solid grounding in Ruby can serve you, as a Rails developer, in four ways:

- By helping you know what the code in your application (including Rails boilerplate code) is doing
- By helping you do more in, and with, your Rails applications than you can if you limit yourself to the readily available Rails idioms and techniques (as powerful as those are)
- By allowing you to familiarize yourself with the Rails source code, which in turn enables you to participate in discussions about Rails and perhaps even submit bug reports and code patches
- By giving you a powerful tool for administrative and organization tasks (for example, legacy code conversion) connected with your application

Downloading Ruby and an Editor

The simplest way to get Ruby installed on a PC is by using the Ruby Installer for Windows. Click on ruby185-22.exe. After you have

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

downloaded this, double-click this file and install Ruby on your PC, accepting all the defaults. After you have installed your Ruby software, the System Environment Variable path is already set to point to the bin folder of Ruby.

The installed Ruby includes the First Edition of Programming Ruby book and the **SciTE** code editor.

Ruby releases with even subversion numbers – 1.6, 1.8, and so on are stable, public releases.

Do note that these instructions assume that you are going to use a Windows platform. For installation on other platforms, you can refer here, here and here.

Ruby Programming Environment

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Let's assume that you have installed Ruby in the folder c:/ruby, then the installation creates a number of sub-folders. The folders:

c:/ruby/bin is where the Ruby executables (including ruby and irb) have been installed.

c:/ruby/lib/ruby/1.8 you'll find program files written in Ruby. These files provide standard library facilities, which you can **require** from your own programs if you need the functionality they provide.

c:/ruby/lib/ruby/1.8/i386-mswin32 contains architecture-specific extensions and libraries. The files in this directory generally have names ending in .so or .dll (depending on your platform). These files are C-language extensions to Ruby; or, more precisely, they are the binary, runtime-loadable files generated from Ruby's C-language extension code, compiled into binary form as part of the Ruby installation process.

c:/ruby/lib/ruby/site_ruby folder is where you and/or your system administrator store third-party extensions and libraries. Some of these may be code you yourself write; others are tools you download from other people's sites and archives of Ruby libraries.

c:/ruby/lib/ruby/gems is the Ruby-Gems packaging system. Inside the gems directory are one or more subdirectories; and if you explore these, you'll

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

find (possibly among other things) the source code for the Rails framework.
c:/ruby/src is where you will find the Ruby source code.
c:/ruby/samples/RubySrc-1.8.4/sample is where you will find some sample Ruby programs.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Our First Ruby program

Let's start our Ruby editor **SciTE**. To do so, on your windows desktop click on *start/Programs/Ruby-185-22/SciTE*. The editor window opens. Press the F8 key to open an output window. Now, click on Options/Open Global Options File and search for 'tabsize'. Edit and make tabsize=2 and indent.size=2. I like my SciTE window to be maximised at start and for that set the position.width=-1 and position.height=-1 Press Ctrl+S and the Ctrl+W. Next, press Ctrl+Shift+I - this opens the Indentation Settings window. Here, ensure that the Tab Size and Indent Size is set to 2 and that the Use tabs box is not checked. Click OK. We are now ready to write our first Ruby program.

Create a folder named say *rubyprograms* on your C:\ We shall store all our programs in this folder. Our first program will display the string 'Hello' on the command window and the name of the program will be p001hello.rb

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

All Ruby source files have the **.rb** file extension. The Ruby coding convention states that file/directory name is lower case of class/module name with .rb extension. For example, Foo class has name foo.rb

In the left window of SciTE type: **puts 'Hello'** and then click *File/Save As...* Give the name p001hello.rb and store it in your *rubyprograms* folder. Press F5 to run your program. You should see Hello in the output window on the right.

Note: In Ruby, program execution proceeds in general from top to bottom. **puts** (s in puts stands for string; **puts** really means put string) simply writes onto the screen whatever comes after it, but then it also automatically goes to the next line.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

In Ruby, everything from an integer to a string is considered to be an object (more on this later). And each object has built in 'methods' (Ruby term for functions) which can be used to do various useful things. To use a method, you need to put a dot after the object, and then append the method name.

Some methods such as **puts** and **gets** are available everywhere and don't need to be associated with a specific object. Technically speaking, these methods are provided by Ruby's **Kernel** module (more on this later) and they are included in all Ruby objects (the **Kernel** module is included by class (more on this later) **Object**, so its methods are available in every Ruby object). When you run a Ruby application, an object called **main** of class **Object** is automatically created. This object provides access to the **Kernel** methods.

The above can be verified by the following program (do not worry if you do not understand the program, right now).

```
puts 'I am an object of class = ' + self.class.to_s
puts 'I am an object = ' + self.to_s
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
print 'The object methods are = '
puts self.private_methods.sort
```

Observe

a. Java and C programmers - no need for a main method/function
b. String literals are sequences of characters between single or double quotation marks. I am using single quotes around Hello. ' is more efficient than " (more on this later)
c. Ruby is an interpreted language, so you don't have to recompile to execute the program written in Ruby
d. Ruby releases with even subversion numbers - 1.6, 1.8, and so on - are stable, public releases
e. The Ruby coding convention states that file/directory name is lower case of class/module name with .rb extension. For example, Foo class has name foo.rb

Some questions asked by you

What do you mean by "Ruby is a Dynamic programming language".

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

In computer science, a dynamic programming language is a kind of programming language in which programs can change their structure as they run: functions may be introduced or removed, new classes of objects may be created, new modules may appear. Refer here for more details.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Some Features of Ruby

Now, let us explore some of the features of Ruby.

1. **Free format**
2. **Case sensitive**
3. **Comments** - Anything following an unquoted **#**, to the end of the line on which it appears, is ignored by the interpreter. Also, to facilitate large comment blocks, the ruby interpreter also ignores anything between a line starting with **=begin** and another line starting with **=end**
4. **Statement delimiters** - Multiple statements on one line must be separated by semicolons, but they are not required at the end of a line; a linefeed is treated like a semicolon. If a line ends with a backslash (\), the linefeed following it is ignored; this allows you to have a single logical line that spans several lines

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Documentation - The complete documentation for Ruby is available online here. The Ruby Cheat Sheet is here.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Numbers in Ruby

Let's play with Numbers. In Ruby, numbers without decimal points are called integers, and numbers with decimal points are usually called floating-point numbers or, more simply, floats (you must place at least one digit before the decimal point). Here's the program p002rubynumbers.rb

```
=begin
  Ruby Numbers
  Usual operators:
  + addition
  - subtraction
  * multiplication
  / division
=end

puts 1 + 2
puts 2 * 3
# Integer division
# When you do arithmetic with integers, you'll get integer answers
puts 3 / 2
puts 10 - 11
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
puts 1.5 / 2.6
```

Ruby integers are objects of class **Fixnum** or **Bignum**. The floating-point numbers are objects of class **Float**, corresponding to the native architecture's double data type.

The class hierarchy is as shown in this figure:

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The above figure courtesy Donald Craig.

Operators and Precedence

Let us look at Ruby's operators. They are arranged here in order from highest to lowest precedence.

::	Scope
[]	Indexing
******	Exponentiation
+ - ! ~	Unary pos/neg, not,...
*** / %**	Multiplication, Division...
+ -	Addition, subtraction...
<< >>	Logical shifts,...
&	Bitwise and
| ^	Bitwise or, xor
> >= < <=	Comparison

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

== === <=> != =~ !~	Equality, inequality...
&&	Boolean and
||	Boolean or
.. ...	Range operators
= (also +=, -=,...)	Assignment
?:	Ternary decision
not	Boolean negation
and, or	Boolean and, or

Assignment: Write a Ruby program that tells you how many minutes are there in a year (do not bother right now about leap years etc.).

Suggestions:

1. Read online Chris Pine's Learn to Program book
2. Read Matz's The Philosophy of Ruby
3. Hear Geoff Grosenbach's Ruby Basics

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Fun with Strings

String literals are sequences of characters between single or double quotation marks.

" (ie. two single quotes) does not have anything in it at all; we call that an empty string.

Here's a program p003rubystrings.rb that explores strings to some extent.

```
=begin
  Ruby Strings
=end
puts "Hello World"
# Can use " or ' for Strings, but ' is more efficient
puts 'Hello World'
# String concatenation with +
puts 'I like' + ' Ruby'
# Escape sequence with \
puts 'It\'s my Ruby'
# displays the string three times
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
puts 'Hello' * 3
```

In Ruby, strings are **mutable**. They can expand as needed, without using much time and memory. Ruby stores a string as a sequence of bytes.

It's worth knowing that a special kind of string exists that uses the back-tick (`) as a beginning and ending delimiter. For example:

```
`dir`
```

The command output string is sent to the operating system as a command to be executed, whereupon the output of the command is substituted back into the string.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Variables and Assignment

To store a number or a string in your computer's memory for use later in your program, you need to give the number or string a name. Programmers often refer to this process as *assignment* and they call the names *variables*. A variable springs into existence as soon as the interpreter sees an assignment to that variable.

```
s = 'Hello World!'
x = 10
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Local variables have the quality of *barewords*; they must start with either a lowercase letter or the underscore character (_), and they must consist entirely of letters, numbers, and underscores. When Ruby sees a plain word sitting there, it interprets it as one of three things: a local variable, a method call, or a keyword.
Keywords (around 38) are special reserved words that you can't use as variable names. def is a keyword; the only thing you can use it for is to start a method definition. if is also a keyword; lots of Ruby code involves conditional clauses that start with if, so it would be too confusing to also allow the use of if as a variable name.
Method calls can be barewords, such as start_here. puts is a method call; so is print. Here's how Ruby decides what it's seeing when it encounters a bareword:

- If there's an equal sign (=) to the right of the bareword, it's a local variable undergoing an assignment.
- If the bareword is a keyword, it's a keyword (Ruby has an internal list of these and recognizes them).
- Otherwise, the bareword is assumed to be a method call.

The p004stringusage.rb program shows us some more usage with strings.

```
# stringusage.rb
# Defining a constant
PI = 3.1416
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
puts PI
# Defining a local variable
my_string = 'I love my city, Pune'
puts my_string
=begin
  The Ruby standard is to use two spaces for indentation
  Conversions
  .to_i, .to_f, .to_s
=end
var1 = 5;
var2 = '2'
# Ruby doesn't automatically convert from strings to integers.
puts var1 + var2.to_i
# << append
a = 'hello'
a<<'world.
I love this world...'
puts a
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

In the example:

```
x = "100".to_i
```

the dot means that the *message* "to_i" is being *sent* to the string "100", or that the method to_i is being called on the string "100". The string "100" is called the *receiver* of the message. Thus, when you see a dot in what would otherwise be an inexplicable position, you should interpret it as a message (on the right) being sent to an object (on the left).

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Scope

Scope refers to the reach or visibility of variables. Different types of variables have different scoping rules. We'll be talking chiefly about two types: *global* and *local* variables.

Global scope and global variables

We're starting with the scope that's used least often, but which you need to be aware of: *global scope*, meaning scope that covers the entire program. Global scope is enjoyed by *global variables*. Global variables are distinguished by starting with a dollar-sign ($) character. They are available everywhere in your program. Global variables never go out of scope. However, global variables are used very little by experienced programmers (except perhaps a few of the built-in ones).

Built-in global variables

The Ruby interpreter starts up with a fairly large number of global variables already initialized. These variables store information that's of

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

potential use anywhere and everywhere in your program. For example, the global variable **$0** contains the name of the file Ruby is executing. The global **$:** (dollar sign followed by a colon) contains the directories that make up the path Ruby searches when you load an external file. **$$** contains the process id of the Ruby process. And there are more.

Local scope

You can tell by looking at a Ruby program where the local scopes begin and end, based on a few rules:

- The top level (outside of all definition blocks) has its own local scope.
- Every class or module definition block (class, module) has its own local scope, *even nested class/module definition blocks.*
- Every method definition (def) has its own local scope.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Getting input

So far we had seen a method like **puts** that writes to the screen. How does one accept user input? For this **gets** (get a string) and **chomp** are useful. The example p005methods.rb below illustrates the same.

```
# gets and chomp
puts "In which city do you stay?"
STDOUT.flush
city = gets.chomp
puts "The city is " + city
```

In the above example when you run it in SciTE, click on the output window and then type the name of your city.

STDOUT is a global constant which is the actual standard output stream for the program. **flush** flushes any buffered data within **io** to the underlying operating system.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

chomp is a string method and **gets** retrieves only strings from your keyboard. You must have realised that **gets** returns a string and a '\n' character, while **chomp** removes this '\n'.

Data comes from many sources. In the typical Rails application, it comes from a database. As a Rails developer, you may find yourself using relatively few of these facilities, because Rails does the data-fetching for you; and your users, when they input from the keyboard, will generally be typing on a Web form.

Assignment:
Write a Ruby program (p006ftoc.rb) that asks for a numeric value of the temperature in degrees Fahrenheit. Finally, the program displays the equivalent value in degrees Centigrade. To format the output to say 2 decimal places, we can use the Kernel's **format** method. For example, if `x = 45.5678` then `format("%.2f", x)` will return a string 45.57. Another way is to use the **round** function as follows `puts (x*100).round/100.0`

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Names in Ruby

Now, let us look at Names in Ruby.

1. **Names** - Ruby names are used to refer to constants, variables, methods, classes, and modules. The first character of a name helps Ruby to distinguish its intended use. Certain names, are reserved words and should not be used as variable, method, class, or module name. Lowercase letter means the characters "a" though "z", as well as "_", the underscore. Uppercase letter means "A" though "Z," and digit means "0" through "9." A name is an uppercase letter, lowercase letter, or an underscore, followed by Name characters: This is any combination of upper- and lowercase letters, underscore and digits.

2. **Variables** - Variables in Ruby can contain data of any type. You can use variables in your Ruby programs without any declarations. Variable name itself denotes its scope (local, global, instance, etc.).

2.1. A **local** variable name consists of a lowercase letter followed by name characters (sunil, _z, hit_and_run).

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

2.2. An **instance** variable (declared within an object) name starts with an "at" sign ("@") followed by an upper- or lowercase letter, optionally followed by name characters (@sign, @_, @Counter).

2.3. A **class** variable (declared within a class) name starts with two "at" signs ("@@") followed by an upper- or lowercase letter, optionally followed by name characters (@@sign, @@_, @@Counter).

2.4. A **constant** name starts with an uppercase letter followed by name characters. Class names and module names are constants, and follow the constant naming conventions. By convention, constant variables are normally spelled using uppercase letters and underscores throughout (module MyMath, PI=3.1416, class MyPune).

2.5. **Global** variables start with a dollar sign ("$") followed by name characters. A global variable name can be formed using "$-" followed by any single character ($counter, $COUNTER, $-x).

3. **Method** names should begin with a lowercase letter. "?" and "!" are the only weird characters allowed as method name suffixes (! or bang labels a method as *dangerous*—specifically, as the dangerous equivalent of a method with the same name but without the bang).

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The Ruby convention is to use underscores to separate words in a multiword method or variable name. For Class names, module names and constants the convention is to use capitalization, rather than underscores, to distinguish the start of words within the name.

It's to be noted that any given variable can at different times hold references to objects of many different types. A Ruby constant is also a reference to an object. Constants are created when they are first assigned to (normally in a class or module definition; they should not be defined in a method - more of this later). Ruby lets you alter the value of a constant, although this will generate a warning message. Also, variables in Ruby act as "references" to objects, which undergo automatic garbage collection.

An example to show Ruby is dynamically typed – p007dt.rb

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
# Ruby is dynamic
x = 7        # integer
x = "house"  # string
x = 7.5      # real
```

The basic types in Ruby are **Numeric** (subtypes include **Fixnum, Integer**, and **Float**), **String, Array, Hash, Object, Symbol, Range**, and **RegEx**. Ruby doesn't require you to use primitives (data types) when manipulating data of these types—if it looks like an integer, it's probably an integer; if it looks like a string, it is probably a string. **class** returns the class of an object, for example:

```
s = 'hello'
s.class  # String
```

In Ruby, everything you manipulate is an object, and the results of those manipulations are themselves objects. There are no primitives or data-types.

```
5.times { puts "Mice!\n" } # more on blocks later
"Elephants Like Peanuts".length
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

More on methods

If objects (such as strings, integers and floats) are the nouns in Ruby language, then methods are the verbs. Every method needs an object. It's usually easy to tell which object is performing the method: it's what comes right before the dot. Sometimes, though, it's not quite as obvious. When we are using **puts**, **gets** - where are their objects? In Ruby, the implicit object is whatever object you happen to be in. But we don't even know how to be in an object yet; we've always been inside a special object (**main**) Ruby has created for us that represents the whole program. You can always see what object you are in (*current object*) by using the special variable **self**.

```
puts self
```

For more details on **self**, refer here.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Writing Own Methods in Ruby

Let's look at writing one's own methods in Ruby with the help of a simple program p008mymethods.rb. Observe that we use **def** and **end** to declare a method. Parameters are simply a list of local variable names in parentheses.

We do not declare the return type; a method returns the value of the last line. It is recommended that you leave a single blank line between each method definition. As per the Ruby convention, methods need parenthesis around their parameters. However, since **puts**, **p** (more on this later) and **gets** are extensively used, the rule of parenthesis is not applicable. In Rails, you will see methods calls with no parentheses.

```
# A simple method
def hello
  puts 'Hello'
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
end
#use the method
hello

# Method with an argument - 1
def hello1(name)
  puts 'Hello ' + name
  return 'success'
end
puts(hello1('satish'))

# Method with an argument - 2
def hello2 name2
  puts 'Hello ' + name2
  return 'success'
end
puts(hello2 'talim') # A method returns the value of the last line
```

The output is:

```
>ruby p008mymethods.rb
Hello
Hello satish
success
Hello talim
success
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
p008mymethods.rb:20: warning: parenthesize argument(s) for future
version
>Exit code: 0
```

Ruby lets you specify default values for a method's arguments—values that will be used if the caller doesn't pass them explicitly. You do this using the assignment operator. See example p009mymethods1.rb

```
def mtd(arg1="Dibya", arg2="Shashank", arg3="Shashank")
  "#{arg1}, #{arg2}, #{arg3}."
end
puts mtd
puts mtd("ruby")
```

The output is:

```
>ruby p009mymethods1.rb
Dibya, Shashank, Shashank.
ruby, Shashank, Shashank.
>Exit code: 0
```

In the above program the interpolation operator #{...} gets calculated separately (please refer to expression interpolation for more details on

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

this), and the results of the calculation are pasted automatically into the string. When you run these lines, you don't see the #{...} operator on your screen; instead, you see the *results* of calculating or *evaluating* what was inside that operator.

The example p010aliasmtd.rb talks about Aliasing a method.

```
alias new_name old_name
```

creates a new name that refers to an existing method. When a method is aliased, the new name refers to a copy of the original method's body. If the method is subsequently redefined, the aliased name will still invoke the original implementation.

```
def oldmtd
  "old method"
end
alias newmtd oldmtd
def oldmtd
  "old improved method"
end
puts oldmtd
puts newmtd
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The output is:

```
>ruby p010aliasmtd.rb
old improved method
old method
>Exit code: 0
```

Does Ruby allow us to write functions that can accept variable number of parameters?

Yes. See the following example p011vararg.rb:

```
def foo(*my_string)
  my_string.each do |words|
    puts words
  end
end
foo('hello','world')
foo()
```

The asterisk is actually taking all arguments you send to the method and assigning them to an array named my_string. The **do end** is a Ruby block

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

which we talk in length here. As you can see, by making use of the asterisk, we're even able to pass in zero arguments. The code above will result in the words hello and world written on successive lines in the first method call and nothing being written on the second call, as you can see in the following output:

```
>ruby p011vararg.rb
hello
world
>Exit code: 0
```

If you want to include optional arguments (*x), they have to come after any non-optional arguments:

```
def opt_args(a,b,*x) # right
def opt_args(a,*x,b) # wrong
```

What is the maximum number of parameters we can pass in Ruby?
There's no limit to the number of parameters. You can refer to: http://www.recentrambles.com/pragmatic/view/68

What is the sequence in which the parameters are put on to the stack? Left to right like C or right to left like Pascal?

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Left to right as you can see in this example p012mtdstack.rb:

```
def mtd(a=99, b=a+1)
  [a,b]
end
puts mtd
```

Are the parameters passed by value or reference?
Look at this url -
http://dev.rubycentral.com/faq/rubyfaq-4.html
and see this example:

```
def downer(string)
  string.downcase!
end
a = "HELLO"     # -> "HELLO"
downer(a)       # -> "hello"
a
```

Bang (!) methods

Ruby methods that modify an object in-place and end in an exclamation mark are known as bang methods. By convention, the bang labels a

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

method as *dangerous*—specifically, as the dangerous equivalent of a method with the same name but without the bang.

You'll find a number of pairs of methods, one with the bang and one without. Those without the bang perform an action and return a freshly minted object, reflecting the results of the action (capitalizing a string, sorting an array, and so on). The bang versions of the same methods perform the action, but they do so *in place*: Instead of creating a new object, they transform the original object.

Examples of such pairs of methods include sort/sort! for arrays, upcase/upcase! for strings, chomp/chomp! for strings, and reverse/reverse! for strings and arrays. In each case, if you call the non-bang version of the method on the object, you get a new object. If you call the bang version, you operate in-place on the same object to which you sent the message.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Ruby Method Missing

When you send a message to an object, the object executes the first method it finds on its method lookup path with the same name as the message. If it fails to find any such method, it raises a **NoMethodError** exception - unless you have provided the object with a method called **method_missing**. The **method_missing** method is passed the symbol of the nonexistent method, and any arguments that were passed in.

method_missing is in part a safety net: It gives you a way to intercept unanswerable messages and handle them gracefully. See the example - p012zmm.rb below (we shall be talking about writing our own class soon).

```
class Dummy
  def method_missing(m, *args)
    puts "There's no method called #{m} here -- please try
again."
  end
end
Dummy.new.anything
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The output is:

```
>ruby tmp.rb
There's no method called anything here -- please try again.
>Exit code: 0
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

More on the String class

There are many methods in the **String** class (you don't have to memorize them all; you can look up the documentation) like the **reverse** that gives a backwards version of a string (**reverse** does not change the original string). **length** that tells us the number of characters (including spaces) in the string. **upcase** changes every lowercase letter to uppercase, and **downcase** changes every uppercase letter to lowercase. **swapcase** switches the case of every letter in the string, and finally, **capitalize** is just like **downcase**, except that it switches the first character to uppercase (if it is a letter), **slice** gives you a substring of a larger string.

The methods **upcase**, **downcase**, **swapcase** and **capitalize** have corresponding methods that modify a string in place rather than creating a new one: **upcase!**, **downcase!**, **swapcase!** and **capitalize!**. Assuming you don't need the original string, these methods will save memory, especially if the string is large.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

We know that String literals are sequences of characters between single or double quotation marks. The difference between the two forms is the amount of processing Ruby does on the string while constructing the literal. In the single-quoted case, Ruby does very little. In the double-quoted case, Ruby does more work. First, it looks for substitutions – sequences that start with a backslash character – and replaces them with some binary value. The second thing that Ruby does with double-quoted strings is expression interpolation. Within the string, the sequence #{expression} is replaced by the value of expression (refer p013expint.rb). In this program, the value returned by a Ruby method is the value of the last expression evaluated, so we can get rid of the temporary variable (result) and the return statement altogether.

```
def say_goodnight(name)
  result = "Good night, #{name}"
  return result
end
puts say_goodnight('Satish')

# modified program
def say_goodnight2(name)
  "Good night, #{name}"
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
puts say_goodnight2('Talim')
```

It is to be noted that every time a string literal is used in an assignment or as a parameter, a new **String** object is created.

Some questions asked by members

How is memory managed for Strings in Ruby? Is there a separate pool for Strings?
Strings are objects of class **String**. The **String** class has more than 75 standard methods. If you refer to Ruby User's Guide, it says that "we do not have to consider the space occupied by a string. We are free from all memory management."

Listing all methods of a class or object

String.methods.sort
shows you a list of methods that the **Class** object **String** responds to.

String.instance_methods.sort

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

This method tells you all the instance methods that instances of **String** are endowed with.

String.instance_methods(false).sort
With this method, you can view a class's instance methods without those of the class's ancestors.

Comparing two strings for equality

Strings have several methods for testing equality. The most common one is == (double equals sign). Another equality-test instance method, **String.eql?**, tests two strings for identical content. It returns the same result as ==. A third instance method, **String.equal?**, tests whether two strings are the same object. An example (p013strcmp.rb) illustrates this:

```
s1 = 'Jonathan'
s2 = 'Jonathan'
s3 = s1
if s1 == s2
  puts 'Both Strings have identical content'
else
  puts 'Both Strings do not have identical content'
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
end
if s1.eql?(s2)
  puts 'Both Strings have identical content'
else
  puts 'Both Strings do not have identical content'
end
if s1.equal?(s2)
  puts 'Two Strings are identical objects'
else
  puts 'Two Strings are not identical objects'
end
if s1.equal?(s3)
  puts 'Two Strings are identical objects'
else
  puts 'Two Strings are not identical objects'
end
```

Exercise

Given a string, let us say that we want to reverse the word order (rather than character order). You can use **String.split**, which gives you an array of words. The **Array** class has a reverse method; so you can reverse the array and **join** to make a new string:

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
words = 'Learning Ruby - Your one stop guide'
puts words.split(" ").reverse.join(" ")
# guide stop one Your - Ruby Learning
```

See the Complete String documentation

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Simple Constructs in Ruby

In Ruby, **nil** and **false** evaluate to false, everything else (including **true**, 0) means true.

Let's explore some very simple constructs available in Ruby. The example below (p014constructs.rb) illustrates the **if else end** construct. By the Ruby convention, **if** and **while** do not require parenthesis.

```
# if else end
var = 5
if var > 4
  puts "Variable is greater than 4"
  puts "I can have multiple statements here"
  if var == 5
    puts "Nested if else possible"
  else
    puts "Too cool"
  end
else
  puts "Variable is not greater than 5"
  puts "I can have multiple statements here"
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

An example of using **elsif** is there in the program p015elsifex.rb as shown below:

```
# elseif example
# Original example
puts "Hello, what\'s your name?"
STDOUT.flush
name = gets.chomp
puts 'Hello, ' + name + '.'

if name == 'Satish'
  puts 'What a nice name!!'
else
  if name == 'Sunil'
    puts 'Another nice name!'
  end
end

# Modified example with elseif
puts "Hello, what\'s your name?"
STDOUT.flush
name = gets.chomp
puts 'Hello, ' + name + '.'

if name == 'Satish'
  puts 'What a nice name!!'
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
elsif name == 'Sunil'
  puts 'Another nice name!'
end

# Further modified
puts "Hello, what\'s your name?"
STDOUT.flush
name = gets.chomp
puts 'Hello, ' + name + '.'

# || is the logical or operator
if name == 'Satish' || name == 'Sunil'
  puts 'What a nice name!!'
end
```

Some common conditional operators are: ==, != >=, <=, >, <

Loops like the **while** loop are available. Again, the example below illustrates the same.

```
# Loops
var = 0
while var < 10
  puts var.to_s
  var += 1
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Assignment:

1. Write a Ruby program (p016leapyear.rb) that asks for a year and then displays to the user whether the year entered by him/her is a *leap year* or not.
2. Write a method **leap_year**. Accept a year value from the user, check whether it's a leap year and then display the number of minutes in that year – program p017leapyearmtd.rb.

Case Expressions

This form is fairly close to a series of **if** statements: it lets you list a series of conditions and execute a statement corresponding to the first one that's true. For example, leap years must be divisible by 400, or divisible by 4 and not by 100. Also, remember that **case** returns the value of the last expression executed.

```
year = 2000
leap = case
       when year % 400 == 0: true
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
       when year % 100 == 0: false
       else year % 4   == 0
       end
puts leap
# output is: true
```

Ruby nil is an Object

In Ruby, **nil** is an actual object. You can call methods on **nil**, just like any other object. You can add methods to **nil**, just like any other object.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Arrays in Ruby

An **Array** is just a list of items in order (like mangoes, apples, and oranges). Every slot in the list acts like a variable: you can see what object a particular slot points to, and you can make it point to a different object. You can make an array by using square brackets and is best explained by the following example p018arrays.rb. Please go through the program carefully.

```
# Arrays

# Empty array
var1 = []
# Array index starts from 0
puts var1[0]

# an array holding a single number
var2 = [5]
puts var2[0]

# an array holding two strings
var3 = ['Hello', 'Goodbye']
puts var3[0]
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
puts var3[1]

flavour = 'mango'
# an array whose elements are pointing
# to three objects - a float, a string and an array
var4 = [80.5, flavour, [true, false]]
puts var4[2]

# a trailing comma is ignored
name = ['Satish', 'Talim', 'Ruby', 'Java',]
puts name[0]
puts name[1]
puts name[2]
puts name[3]
# the next one outputs nil
# nil is Ruby's way of saying nothing
puts name[4]
# we can add more elements too
name[4] = 'Pune'
puts name[4]
# we can add anything!
name[5] = 4.33
puts name[5]
# we can add an array to an array
name[6] = [1, 2, 3]
puts name[6]

# some methods on arrays
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
newarr = [45, 23, 1, 90]
puts newarr.sort
puts newarr.length

# method each (iterator) - extracts each element into lang
languages = ['Pune', 'Mumbai', 'Bangalore']

languages.each do |lang|
  puts 'I love ' + lang + '!'
  puts 'Don\'t you?'
end

# delete an entry in the middle and shift the remaining entries
languages.delete('Mumbai')
languages.each do |lang|
  puts 'I love ' + lang + '!'
  puts 'Don\'t you?'
end
```

The method **each** allows us to do something (whatever we want) to each object the array points to. In the example, we are able to go through each object in the array without using any numbers.

Here are a few things to remember:

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

- The variable lang inside the "goalposts" refers to each item in the array as it goes through the loop. You can give this any name you want, but make it memorable.
- The **do** and **end** identify a block of code that will be executed for each item. Blocks are used extensively in Ruby and we shall spend more time on them later.

Here's an interesting example of a method that returns an array.
Example p019mtdarry.rb

```
def mtdarry
  10.times do |num|
    square = num * num
    return num, square if num > 5
  end
end

num, square = mtdarry
puts num
puts square
```

The output is:

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
>ruby mtdarry.rb
6
36
>Exit code: 0
```

The **times** method of the **Integer** class iterates block num times, passing in values from zero to num-1. As we can see, if you give **return** multiple parameters, the method returns them in an array. You can use parallel assignment to collect this return value.

Some questions asked by you

Does Ruby have associative arrays like awk?
Hashes (sometimes known as associative arrays, maps or dictionaries) are available in Ruby.

See the complete Array documentation

Assignment:

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

1. Write a Ruby program (p020arraysum.rb) that, when given an array as collection = [1, 2, 3, 4, 5] it calculates the sum of its elements.

2. Write a Ruby program (p021oddeven.rb) that, when given an array as collection = [12, 23, 456, 123, 4579] it displays for each number, whether it is odd or even.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Ranges

The first and perhaps most natural use of ranges is to express a sequence. Sequences have a start point, an end point, and a way to produce successive values in the sequence. In Ruby, these sequences are created using the ". ." and ". . ." range operators. The two dot form creates an inclusive range, and the three-dot form creates a range that excludes the specified high value. In Ruby ranges are not represented internally as lists: the sequence 1..100000 is held as a **Range** object containing references to two **Fixnum** objects. Refer program p021ranges.rb. If you need to, you can convert a range to a list using the **to_a** method.

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

Ranges implement methods that let you iterate over them and test their contents in a variety of ways.

```
digits = 0..9
digits.include?(5)          -> true
digits.min                  -> 0
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
digits.max                    -> 9
digits.reject {|i| i < 5 }  -> [5, 6, 7, 8, 9]
```

Another use of the versatile range is as an interval test: seeing if some value falls within the interval represented by the range. We do this using ===, the case equality operator.

```
(1..10) === 5       -> true
(1..10) === 15      -> false
(1..10) === 3.14159 -> true
('a'..'j') === 'c'  -> true
('a'..'j') === 'z'  -> false
```

Assignment: Given a string **s = 'key=value'**, create two strings s1 and s2 such that s1 contains key and s2 contains value. Hint: Use some of the **String** functions – program p021rangesex.rb

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Blocks and Procs

Ruby Code blocks (called *closures* in other languages) are definitely one of the coolest features of Ruby and are chunks of code between braces or between **do**… **end** that you can associate with method invocations, almost as if they were parameters. A Ruby block is a way of grouping statements, and may appear only in the source adjacent to a method call; the block is written starting on the same line as the method call's last parameter (or the closing parenthesis of the parameter list). The code in the block is not executed at the time it is encountered. Instead, Ruby remembers the context in which the block appears (the local variables, the current object, and so on) and then enters the method.

The Ruby standard is to use braces for single-line blocks and **do**… **end** for multi-line blocks. Keep in mind that the braces syntax has a higher precedence than the **do..end** syntax.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Matz says that any method can be called with a block as an implicit argument. Inside the method, you can call the block using the **yield** keyword with a value.

Once you have created a block, you can associate it with a call to a method. Usually the code blocks passed into methods are anonymous objects, created on the spot. For example, in the following code, the block containing **puts** "Hello" is associated with the call to a method greet.

```
greet {puts 'Hello'}
```

If the method has parameters, they appear before the block.

```
verbose_greet("PuneRuby") {puts 'Hello'}
```

A method can then invoke an associated block one or more time using the Ruby **yield** statement.

Program p022codeblock.rb illustrates what we have just discussed.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
def call_block
  puts 'Start of method'
  yield
  yield
  puts 'End of method'
end
call_block {puts 'In the block'}
```

The output is:

```
>ruby codeblock.rb
Start of method
In the block
In the block
End of method
>Exit code: 0
```

If you provide a code block when you call a method, then *inside the method*, you can yield control to that code block—suspend execution of the method; execute the code in the block; and return control to the method body, right after the call to **yield**.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

You can provide parameters to the call to **yield**: these will be passed to the block. Within the block, you list the names of the arguments to receive the parameters between vertical bars (|).

The example p023codeblock2.rb illustrates the same.

```
def call_block
  yield('hello', 99)
end
call_block {|str, num| puts str + ' ' + num.to_s}
```

The Output is:

```
>ruby codeblock2.rb
hello 99
>Exit code: 0
```

Note that the code in the block is not executed at the time it is encountered by the Ruby interpreter. Instead, Ruby remembers the context in which the block appears and then enters the method.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

A code block's return value (like that of a method) is the value of the last expression evaluated in the code block. This return value is made available inside the method; it comes through as the return value of yield.

Blocks are not objects, but they can be converted into objects of class **Proc**. This can be done by calling the **lambda** method of the module **Kernel**. A block created with **lambda** acts like a Ruby method. If you don't specify the right number of arguments, you can't call the block.

```
prc = lambda {"hello"}
```

Proc objects are blocks of code that have been bound to a set of local variables. The class **Proc** has a method **call** that invokes the block. The program p024proccall.rb illustrates this.

```
prc = lambda {puts 'Hello'}
prc.call

# another example
toast = lambda do
  puts 'Cheers'
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
end
toast.call
```

The output is:

```
>ruby proccall.rb
Hello
Cheers
>Exit code: 0
```

Remember you cannot pass methods into other methods (but you can pass procs into methods), and methods cannot return other methods (but they can return procs).

The next example shows how methods can take procs. Example p025mtdproc.rb

```
def some_mtd some_proc
  puts 'Start of mtd'
  some_proc.call
  puts 'End of mtd'
end

say = lambda do
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
  puts 'Hello'
end

some_mtd say
```

The output is:

```
>ruby mtdproc.rb
Start of mtd
Hello
End of mtd
>Exit code: 0
```

Here's another example of passing arguments using **lambda**.

```
aBlock = lambda { |x| puts x }
aBlock.call 'Hello World!'
# output is: Hello World!
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Random Numbers

Ruby comes with a random number generator. The method to get a randomly chosen number is **rand**. If you call **rand**, you'll get a float greater than or equal to 0.0 and less than 1.0. If you give it an integer parameter (by calling **rand**(5), you will get an integer value greater than or equal to 0 and less than 5.

Here's an example: p026phrase.rb

```
word_list_one = ['24/7', 'multi-Tier', '30,000 foot', 'B-to-B',
'win-win', 'front-end', 'web-based', 'pervasive', 'smart', 'six-
sigma', 'critical-path', 'dynamic']
word_list_two = ['empowered', 'sticky', 'value-added', 'oriented',
'centric', 'distributed', 'clustered', 'branded', 'outside-the-
box', 'positioned', 'networked', 'focused', 'leveraged',
'aligned', 'targeted', 'shared', 'cooperative', 'accelerated']
word_list_three = ['process', 'tipping-point', 'solution',
'architecture', 'core competency', 'strategy', 'mindshare',
'portal', 'space', 'vision', 'paradigm', 'mission']

one_len = word_list_one.length
two_len = word_list_two.length
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
three_len = word_list_three.length

rand1 = rand(one_len)
rand2 = rand(two_len)
rand3 = rand(three_len)

phrase = word_list_one[rand1] + " " + word_list_two[rand2] + " " +
word_list_three[rand3]

puts phrase
```

The above program makes three lists of words, and then randomly picks one word from each of the three lists and prints out the result.

Assignment:
This assignment is from Chris Pine's Book.

a. Write a Deaf Grandma program (p026zdeafgm1.rb). Whatever you say to grandma (whatever you type in), she should respond with HUH?! SPEAK UP, SONNY!, unless you shout it (type in all capitals). If you shout, she can hear you (or at least she thinks so) and yells back, NO, NOT SINCE 1938! To make your program really believable, have grandma shout a different year each time;

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

maybe any year at random between 1930 and 1950. You can't stop talking to grandma until you shout BYE.

b. Program (p026zdeafgm2.rb) - Extend your Deaf Grandma program: What if grandma doesn't want you to leave? When you shout BYE, she could pretend not to hear you. Change your previous program so that you have to shout BYE three times in a row. Make sure to test your program: if you shout BYE three times, but not in a row, you should still be talking to grandma.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Reading from / Writing to text files

Let's look at how we can read / write to a text file with the help of a simple program p027readwrite.rb.

```
# Open and read from a text file
File.open('Constructs.rb', 'r') do |f1|
 while line = f1.gets
  puts line
 end
end

# Create a new file and write to it
File.open('Test.rb', 'w') do |f2|
 # use "" for inserting a newline between quotation marks
 f2.puts "Created by Satish\nThank God!"
end
```

The **File.open** method can open the file in different modes like 'r' Read-only, starts at beginning of file (default); 'r+' Read/Write, starts at beginning of file; 'w' Write-only, truncates existing file to zero length or creates a new file for writing. Please check the online documentation

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

for a full list of modes available. **File.open** opens a new File if there is no associated block. If the optional block is given, it will be passed file as an argument, and the file will automatically be closed when the block terminates.

Assignment: Write a Ruby program (call it p028swapcontents.rb) to do the following. Take two text files say A and B. The program should swap the contents of A and B ie. after the program is executed, A should contain B's contents and B should contains A's.

Traversing Directory Trees

The Find module supports top-down traversal of a set of file paths, given as arguments to the find method. If an argument is a directory, then its name and name of all its files and sub-directories will be passed in (in the example below, this would be from where you run this program).

```
require 'find'

Find.find('./') do |f|
  type = case
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
        when File.file?(f): "F"
        when File.directory?(f): "D"
        else "?"
        end
  puts "#{type}: #{f}"
end
```

We shall talk about **require** soon here.

Random Access

It's quite easy to access a file randomly. Let's say we have a text file (named **hellousa.rb**) , the contents of which is shown below:

```
puts 'Hello USA'
```

We now need to display the contents of the file from the word USA. Here's how – program p028xrandom.rb:

```
f = File.new("hellousa.rb")
f.seek(12, IO::SEEK_SET)
print f.readline
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The **seek** method of class **IO**, seeks to a given offset *anInteger* (first parameterof method) in the stream according to the value of second parameter in the method.

```
IO::SEEK_CUR  | Seeks to _amount_ plus current position
--------------+-----------------------------------------------
IO::SEEK_END  | Seeks to _amount_ plus end of stream (you
              | probably
              | want a negative value for _amount_)
--------------+-----------------------------------------------
IO::SEEK_SET  | Seeks to the absolute location given by
              | _amount_
```

More on the scope operator **::** here.

Some questions asked by you

Does Ruby allow Object Serialization?

Java features the ability to serialize objects, letting you store them somewhere and reconstitute them when needed. Ruby calls this kind of serialization marshaling. Saving an object and some or all of its components is done using the method **Marshal.dump**. Later on you can

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

reconstitute the object using **Marshal.load**. Ruby uses marshaling to store session data. Refer topic Object Serialization later on.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Writing our own Class

Objects are your agents, your proxies, in the universe of your program. You ask them for information. You assign them tasks to accomplish. You tell them to perform calculations and report back to you. You hand them to each other and get them to work together.

When you design a class, think about the objects that will be created from that class type. Think about the things the object *knows* and the things the object *does*.

Things an object *knows* about itself are called instance variables. They represent an object's state (the data - for example, the quantity and the product id), and can have unique values for each object of that type.

Things an object can *do* are called methods.

Thus, an object is an entity that serves as a container for data and also controls access to the data. Associated with an object is a set of

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

attributes, which are essentially no more than variables belonging to that object. Also associated with an object is a set of functions that provide an interface to the functionality of the object, called methods.

A class is a combination of state and methods that use the state.

Hence a class is used to construct an object. A class is a blueprint for an object. For example, you might use a Button class to make dozens of different buttons, and each button might have its own color, size, shape, label, and so on.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Read this very carefully, it's a brain bender!
Classes in Ruby are first-class objects - each is an instance of class **Class**. When a new class is defined (typically using **class** Name ... **end**), an object of type **Class** is created and assigned to a constant (Name. in this case). When Name.**new** is called to create a new object, the **new** instance method in **Class** is run by default, which in turn invokes **allocate** to allocate memory for the object, before finally calling the new object's **initialize** method. The constructing and initializing phases of an object are separate and both can be over-ridden. The initialization is done via the **initialize** instance method while the construction is done via the **new** class method. **initialize** is not a constructor!

The following Class Hierarchy is informative.

Let's write our first, simple class – p029dog.rb

```
# define class Dog
class Dog
  def initialize(breed, name)
    # Instance variables
    @breed = breed
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
    @name = name
  end

  def bark
    puts 'Ruff! Ruff!'
  end

  def display
    puts "I am of #{@breed} breed and my name is #{@name}"
  end
end

# make an object
# method new used to create object
d = Dog.new('Labrador', 'Benzy')
puts d.methods.sort
puts "The id of obj is #{d.object_id}."
if d.respond_to?("talk")
  d.talk
else
  puts "Sorry, the object doesn't understand the 'talk' message."
end

d.bark
d.display

d1 = d
d1.display
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
d = nil
d.display
```

The output is:

```
>ruby dog.rb
Ruff! Ruff!
I am of Labrador breed and my name is Benzy
>Exit code: 0
```

The method **new** is used to create an object of class Dog. Objects are created on the heap. The variable d is known as a reference variable. It does not hold the object itself, but it holds something like a pointer or an address of the object. You use the dot operator (.) on a reference variable to say, "use the thing *before* the dot to get me the thing *after* the dot." For example:

d.bark

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

If you're writing a Rails application in which one of your entity models is, say, *Customer*, then when you write the code that causes things to happen - a customer logging into a site, updating a customer's phone number, adding an item to a customer's shopping cart - in all likelihood you'll be sending messages to customer objects.

Even a newly created object isn't a blank slate. As soon as an object comes into existence, it already responds to a number of messages. Every object is "born" with certain innate abilities. To see a list of innate methods, you can call the **methods** method (and throw in a **sort** operation, to make it easier to browse visually):

puts d.methods.sort

The result is a list of all the messages (methods) this newly minted object comes bundled with. Amongst these many methods, the methods **object_id** and **respond_to?** are important.

Every object in Ruby has a unique id number associated with it. You can see an object's id by asking the object to show you its **object_id**:

```
puts "The id of obj is #{d.object_id}."
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

You can determine in advance (before you ask the object to do something) whether the object knows how to handle the message you want to send it, by using the **respond_to?** method. This method exists for all objects; you can ask any object whether it responds to any message. **respond_to?** usually appears in connection with conditional (if) logic.

```
if d.respond_to?("talk")
  d.talk
else
  puts "Sorry, the object doesn't understand the 'talk' message."
end
```

Now, the statements:

```
d1 = d
d1.display
```

makes d and d1 point to the same object.

Literal Constructors

That means you can use special notation, instead of a call to **new**, to create a new object of that class. The classes with literal constructors are shown in the table below. When you use one of these literal constructors, you bring a new object into existence.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Class	Literal constructor	Example(s)
String	Quotation marks	`"new string"` or `'new string'`
Symbol	Leading colon	`:symbol` or `:"symbol with spaces"`
Array	Square brackets	`[1,2,3,4,5]`
Hash	Curly braces	`{"New York" => "NY", "Oregon" => "OR"}`
Range	Two or three dots	`0...10` or `0..9`
Regexp	Forward slashes	`/([a-z]+)/`

Garbage Collection

The statement:
d =nil
makes d a **nil** reference, meaning it does not refer to anything. If I now say:
d1 = nil

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

then the Dog object is abandoned and eligible for Garbage collection. The Ruby object heap allocates a minimum of 8 megabytes. Ruby's GC is called mark-and-sweep. The "mark" stage checks objects to see if they are still in use. If an object is in a variable that can still be used in the current scope, the object (and any object inside that object) is marked for keeping. If the variable is long gone, off in another method, the object isn't marked. The "sweep" stage then frees objects which haven't been marked. Ruby uses a conservative mark-and-sweep GC mechanism. There is no guarantee that an object will undergo garbage collection before the program terminates.

If you stuff something in an array and you happen to keep that array around, **it's all marked**. If you stuff something in a constant or global variable, **it's forever marked**.

Class Methods

The idea of a class method is that you send a message to the object that is the class rather than to one of the class's instances. Class methods serve a purpose. Some operations pertaining to a class can't be

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

performed by individual instances of that class. **new** is an excellent example. We call **Dog.new** because, until we've created an individual dog instance, we can't send it *any* messages! Besides, the job of spawning a new object logically belongs to the class. It doesn't make sense for instances of **Dog** to spawn each other. It does make sense, however, for the instance-creation process to be centralized as an activity of the class Dog. It's vital to understand that by **Dog.new**, we have a method that we can access through the class object **Dog** *but not through its instances.* Individual dog objects (instances of the class **Dog**) *do not have this method.* A class object (like **Dog**) has its own methods, its own state, its own identity. It doesn't share these things with instances of itself.

The table below shows an example (borrowed from the book Ruby For Rails) on the notation used:

Notation	Method referred to
Ticket#price	The instance method `price` in the class `Ticket`
Ticket.most_expensive	The class method `most_expensive`, in the class `Ticket`
Ticket::most_expensive	Another way to refer to the class method `most_expensive`

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

You will learn how to write class methods, later on here.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Including Other Files

When writing your first Ruby programs, you tend to place all of your code in a single file. But as you grow as a Ruby programmer, your Ruby programs will also grow, and at some point you will realize that having a single file contain all of your code just won't do. It is easier to break your code up into logical groupings and place each group in a separate file or files. When you begin using multiple files, you have a need for the Ruby **require** and **load** methods that help you include other files in your program.

The **load** method includes the named Ruby source file every time the method is executed:

load 'filename.rb'

The more commonly used **require** method loads any given file only once:

require 'filename'

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

require gives you access to the many *extensions* and *programming libraries* bundled with the Ruby programming language—as well as an even larger number of extensions and libraries written independently by other programmers and made available for use with Ruby.

Note that you say **require 'filename'**, not **require 'filename.rb'**. Aside from looking nicer, this bareword way of referring to the extension is necessary because not all extensions use files ending in .rb. Specifically, extensions written in C are stored in files ending with .so or .dll. To keep the process transparent—that is, to save you the trouble of knowing whether the extension you want uses a .rb file or not—Ruby accepts a bareword and then does some automatic file-searching and trying out of possible filenames until it finds the file corresponding to the extension you have requested.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Rails uses **load** in preference to **require**, for example, in development mode - which means that if you're trying your application in a browser and making changes to the code at the same time, your changes are reloaded, overriding any caching behavior on the part of the Web server. Multiple **require** calls in the same place don't have the same effect if the application has already read the file in once.

Now, let's look at an example of another class - p030motorcycle.rb.

```
class MotorCycle
  def initialize(make, color)
    # Instance variables
    @make = make
    @color = color
    end
  def startEngine
    if (@engineState)
      puts 'Engine Running'
    else
      @engineState = true
      puts 'Engine Idle'
    end
  end
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

We write another program p031motorcycletest.rb to test out the above class.

```
require 'p030motorcycle'
m = MotorCycle.new('Yamaha', 'red')
m.startEngine
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Open classes

In Ruby, classes are never closed: you can always add methods to an existing class. This applies to the classes you write as well as the standard, built-in classes. All you have to do is open up a class definition for an existing class, and the new contents you specify will be added to whatever's there.

Now to the above program p031motorcycletest.rb add the method dispAttr

```
require 'motorcycle'
m = MotorCycle.new('Yamaha', 'red')
m.startEngine

class MotorCycle
  def dispAttr
    puts 'Color of MotorCycle is ' + @color
    puts 'Make  of MotorCycle is ' + @make
  end
end
m.dispAttr
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
m.startEngine
puts self.class
puts self
```

Please note that **self.class** refers to and **self** refers to an object called **main** of class **Object**.

One more example is program - p031xdognext.rb

```
require 'p029dog.rb'
# define class Dog
class Dog
  def big_bark
    puts 'Woof! Woof!'
  end
end
# make an object
d = Dog.new('Labrador', 'Benzy')
d.bark
d.big_bark
d.display
```

Here's another example of adding a method to the **String** class. The program p032mystring.rb illustrates the same.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
class String
  def writesize
    puts self.size
  end
end
size_writer = "Tell me my size!"
size_writer.writesize
```

(You can confirm the output to the above programs yourself).

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Inheritance

Inheritance is a relation between two classes. We know that all cats are mammals, and all mammals are animals. Smaller classes inherit characteristics from the larger classes to which they belong. If all mammals breathe, then all cats breathe.

We can express this concept in Ruby - see the p033mammal.rb program below:

```
class Mammal
  def breathe
    puts "inhale and exhale"
  end
end

class Cat<Mammal
  def speak
    puts "Meow"
  end
end

rani = Cat.new
rani.breathe
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
rani.speak
```

Though we didn't specify how a Cat should breathe, every cat will inherit that behaviour from the Mammal class since Cat was defined as a subclass of Mammal. (In OO terminology, the smaller class is a subclass and the larger class is a super-class. The subclass is sometimes also known as a derived or child class and the super-class as base or parent class). Hence from a programmer's standpoint, cats get the ability to breathe for free; after we add a speak method, our cats can both breathe and speak.

There will be situations where certain properties of the super-class should not be inherited by a particular subclass. Though birds generally know how to fly, penguins are a flightless subclass of birds. In the example p034bird.rb below, we *override* fly in class Penguin.

```
class Bird
  def preen
    puts "I am cleaning my feathers."
  end
  def fly
    puts "I am flying."
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
  end
end

class Penguin<Bird
  def fly
    puts "Sorry. I'd rather swim."
  end
end

p = Penguin.new
p.preen
p.fly
```

Rather than exhaustively define every characteristic of every new class, we need only to append or to redefine the differences between each subclass and its super-class. This use of inheritance is sometimes called differential programming. It is one of the benefits of object-oriented programming.

The above two programs are taken from the online Ruby User's Guide.

Thus, Inheritance allows you to create a class that is a refinement or specialization of another class. Inheritance is indicated with **<**.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Here's another example, p035inherit.rb.

```
class GF
 def initialize
  puts 'In GF class'
 end
 def gfmethod
  puts 'GF method call'
 end
end

# class F sub-class of GF
class F < GF
 def initialize
  puts 'In F class'
 end
end

# class S sub-class of F
class S < F
 def initialize
  puts 'In S class'
 end
end
son = S.new
son.gfmethod
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

A class can only inherit from one class at a time (i.e. a class can inherit from a class that inherits from another class which inherits from another class, but a single class can not inherit from many classes at once).

There are many classes and modules (more on this later) built into the standard Ruby language. They are available to every Ruby program automatically; no **require** is required. Some built-in classes are **Array, Bignum, Class, Dir, Exception, File, Fixnum, Float, Integer, IO, Module, Numeric, Object, Range, String, Thread, Time**. Some built-in modules are **Comparable, Enumerable, GC, Kernel, Math**.

The **Object** class is the parent class of all classes in Ruby. Its methods are therefore available to all objects unless explicitly overridden.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Inheritance is one of the key organizational techniques for Rails program design and the design of the Rails framework.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Duck Typing

You'll have noticed that in Ruby we don't declare the types of variables or methods - everything is just some kind of object. Ruby objects (unlike objects in some other object-oriented languages) can be individually modified. You can always add methods on a per object basis. In Ruby, the behavior or capabilities of an object can deviate from those supplied by its class.

In Ruby, we rely less on the type (or class) of an object and more on its capabilities. Hence, Duck Typing means an object type is defined by what it can do, not by what it is. Duck Typing refers to the tendency of Ruby to be less concerned with the class of an object and more concerned with what methods can be called on it and what operations can be performed on it. In Ruby, we would use **respond_to?** or might simply pass an object to a method and know that an exception will be raised if it is used inappropriately.

If an object walks like a duck and talks like a duck, then the Ruby interpreter is happy to treat it as if it were a duck.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Consider the following example.

```
# Check whether the object defines the to_str method
puts ('A string'.respond_to? :to_str) # => true
puts (Exception.new.respond_to? :to_str) # => true
puts (4.respond_to? :to_str) # => false
```

The above example is the simplest example of Ruby's philosophy of "duck typing:" if an object quacks like a duck (or acts like a string), just go ahead and treat it as a duck (or a string). Whenever possible, you should treat objects according to the methods they define rather than the classes from which they inherit or the modules they include.

Exceptions, for instance, are essentially strings that have extra information associated with them. But they don't subclass class name "String". Code that uses **is_a?** String to check for stringness will overlook the essential stringness of Exceptions.

Now consider the following three classes – Duck, Goose and DuckRecording. Program p036duck.rb

```
class Duck
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
  def quack
    'Quack!'
  end

  def swim
    'Paddle paddle paddle...'
  end
end

class Goose
  def homk
    'Hock!'
  end
  def swim
    'Splash splash splash...'
  end
end

class DuckRecording
  def quack
    play
  end

  def play
    'Quack!'
  end
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
def make_it_quack(duck)
  duck.quack
end
puts make_it_quack(Duck.new)
puts make_it_quack(DuckRecording.new)

def make_it_swim(duck)
  duck.swim
end
puts make_it_swim(Duck.new)
puts make_it_swim(Goose.new)
```

If you refer to the code shown below:

```
def make_it_quack(duck)
  duck.quack
end
puts make_it_quack(Duck.new)
puts make_it_quack(DuckRecording.new)
```

A method that told a Duck to quack works when given a DuckRecoding, due to Duck Typing. Similarly in the following code:

```
def make_it_swim(duck)
  duck.swim
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
puts make_it_swim(Duck.new)
puts make_it_swim(Goose.new)
```

A method that tells a Duck to swim when given a Goose, works.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Overloading Methods

You want to create two different versions of a method with the same name: two methods that differ in the arguments they take. However, a Ruby class can have only one method with a given name. Within that single method, though, you can put logic that branches depending on how many and what kinds of objects were passed in as arguments.

Here's a *Rectangle* class that represents a rectangular shape on a grid. You can instantiate a *Rectangle* by one of two ways: by passing in the coordinates of its top-left and bottom-left corners, or by passing in its top-left corner along with its length and width. There's only one **initialize** method, but you can act as though there were two.

```
# The Rectangle initialize accepts arguments in either
# of the following forms:
#   Rectangle.new([x_top, y_left], length, width)
#   Rectangle.new([x_top, y_left], [x_bottom, y_right])
class Rectangle
  def initialize(*args)
    if args.size < 2  || args.size > 3
      # modify this to raise exception, later
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
      puts 'This method takes either 2 or 3 arguments'
    else
      if args.size == 2
        puts 'Two arguments'
      else
        puts 'Three arguments'
      end
    end
  end
end
Rectangle.new([10, 23], 4, 10)
Rectangle.new([10, 23], [14, 13])
```

The above code (p037rectangle.rb) is incomplete from the *Rectangle* class viewpoint, but is enough to demonstrate how method overloading can be achieved. Also remember that the **initialize** method takes in a variable number of arguments.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Overriding Methods

Method overriding, in object oriented programming, is a language feature that allows a subclass to provide a specific implementation of a method that is already provided by one of its super-classes. The implementation in the subclass overrides (replaces) the implementation in the super-class.

Here's an example - p037xmtdovride.rb:

```
class A
  def a
    puts 'In class A'
  end
end

class B < A
  def a
    puts 'In class B'
  end
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
b = B.new
b.a
```

The method a in class B overrides the method a in class A.

Usage of super

The way **super** handles arguments, is as follows:

- When you invoke **super** with no arguments Ruby sends a message to the parent of the current object, asking it to invoke a method of the same name as the method invoking **super**. It automatically forwards the arguments that were passed to the method from which it's called.
- Called with an empty argument list - **super**()—it sends *no* arguments to the higher-up method, even if arguments were passed to the current method.
- Called with specific arguments - **super**(a, b, c)—it sends exactly those arguments.

An example (p038bicycle.rb) from Ruby for Rails book highlights this:

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
class Bicycle
  attr_reader :gears, :wheels, :seats
  def initialize(gears = 1)
    @wheels = 2
    @seats = 1
    @gears = gears
  end
end

class Tandem < Bicycle
  def initialize(gears)
    super
    @seats = 2
  end
end
t = Tandem.new(2)
puts t.gears
puts t.wheels
puts t.seats
b = Bicycle.new
puts b.gears
puts b.wheels
puts b.seats
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The output is:

```
>ruby p038bicycle.rb
2
2
2
1
2
1
>Exit code: 0
```

Redefining methods

(Adapted from David Black's book - Ruby For Rails)

Nothing stops you from defining a method twice. Program p038or.rb

```
class OR
  def mtd
    puts "First definition of method mtd"
  end
  def mtd
    puts "Second definition of method mtd"
  end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
end
OR.new.mtd
```

What happens when we call mtd on an instance of OR? Let's find out:

```
OR.new.mtd
```

The printed result is Second definition of method mtd. The second definition has prevailed: We see the output from that definition, not from the first. **When you *override* a method, the new version takes precedence**.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Symbols

A symbol looks like a variable name but it's prefixed with a colon. Examples - **:action**, **:line_items**. You can think of symbols as string literals that are magically turned into constants. Alternatively, you can consider the colon to mean "thing named" so **:id** is "the thing named id." You can also think of **:id** as meaning the name of the variable id, and plain id as meaning the value of the variable.

A **Symbol** is the most basic Ruby object you can create. It's just a name and an internal ID. Symbols are useful because a given symbol name refers to the same object throughout a Ruby program. Symbols are more efficient than strings. Two strings with the same contents are two different objects, but for any given name there is only one **Symbol** object. This can save both time and memory.

See the example: p039symbol.rb

```
# p039symbol.rb
# use the object_id method of class Object
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
# it returns an integer identifier for an object
puts "string".object_id
puts "string".object_id
puts :symbol.object_id
puts :symbol.object_id
```

The output when I ran the program on my PC was:

```
>ruby p039symbol.rb
21066960
21066930
132178
132178
>Exit code: 0
```

Therefore, when do we use a string versus a symbol?

- If the contents (the sequence of characters) of the object are important, use a string
- If the identity of the object is important, use a symbol

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Ruby uses symbols, and maintains a Symbol Table to hold them. Symbols are names - names of instance variables, names of methods, names of classes. So if there is a method called control_movie, there is automatically a symbol :control_movie. Ruby's interpreted, so it keeps its Symbol Table handy at all times. You can find out what's on it at any given moment by calling **Symbol.all_symbols**

A Symbol object is created by prefixing an operator, string, variable, constant, method, class, module name with a colon. The symbol object will be unique for each different name but does not refer to a particular instance of the name, for the duration of a program's execution. Thus, if Fred is a constant in one context, a method in another, and a class in a third, the Symbol :Fred will be the same object in all three contexts.

This can be illustrated by this simple program – p039xsymbol.rb:

```
class Test
  puts :Test.object_id.to_s
  def test
    puts :test.object_id.to_s
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
    @test = 10
    puts :test.object_id.to_s
  end
end
t = Test.new
t.test
```

The output is:

```
>ruby p039xsymbol.rb
116458
79218
79218
>Exit code: 0
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Hashes

Hashes (sometimes known as *associative arrays*, *maps*, or *dictionaries*) are similar to arrays in that they are indexed collection of object references. However, while you index arrays with integers, you can index a hash with objects of any types: strings, regular expressions, and so on. When you store a value in a hash, you actually supply two objects – the index (normally called the *key*) and the value. You can subsequently retrieve the value by indexing the hash with the same key. The values in a hash can be objects of any type.

The example p040myhash.rb that follows uses hash literals: a list of *key* => *value* pairs between braces.

```
h = {'dog' => 'canine', 'cat' => 'feline', 'donkey' => 'asinine',
12 => 'dodecine'}
puts h.length  # 3
puts h['dog']   # 'canine'
puts h
puts h[12]
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The output is:

```
>ruby p040myhash.rb
4
canine
catfeline12dodecinedonkeyasininedogcanine
dodecine
>Exit code: 0
```

Compared with arrays, hashes have one significant advantage: they can use any object as an index. However, their elements are not ordered, so you cannot easily use a hash as a stack or a queue.

Hashes have a *default value*. This value is returned when an attempt is made to access keys that do not exist in the hash. By default this value is **nil**.

The **Hash** class has many methods and you can refer them here.

Using Symbols as Hash Keys

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Whenever you would otherwise use a quoted string, use a symbol instead. See the following example p041symbolhash.rb

```
people = Hash.new
people[:nickname] = 'IndianGuru'
people[:language] = 'Marathi'
people[:lastname] = 'Talim'

puts people[:lastname] # Talim
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Exploring Time class

The **Time** class in Ruby has a powerful formatting function which can help you represent the time in a variety of ways. The **Time** class contains Ruby's interface to the set of time libraries written in C. Time zero for Ruby is the first second GMT of January 1, 1970. Ruby's **DateTime** class is superior to **Time** for astronomical and historical applications, but you can use **Time** for most everyday programs.

The **strftime** function is modelled after C's printf. The p042time.rb program shows some of these functions.

```
# First get the current local time
t = Time.now
# to get day, month and year with century
# also hour, minute and second
puts t.strftime("%d/%m/%Y %H:%M:%S")

# You can use the upper case A and B to get the full
# name of the weekday and month, respectively
puts t.strftime("%A")
puts t.strftime("%B")
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
# You can use the lower case a and b to get the abbreviated
# name of the weekday and month, respectively
puts t.strftime("%a")
puts t.strftime("%b")

# 24 hour clock and Time zone name
puts t.strftime("at %H:%M %Z")
```

The output is:

```
>ruby p042time.rb
10/09/2006 10:06:31
Sunday
September
Sun
Sep
at 10:06 India Standard Time
>Exit code: 0
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Exceptions

An *exception* is a special kind of object, an instance of the class **Exception** or a descendant of that class. *Raising* an exception means stopping normal execution of the program and either dealing with the problem that's been encountered or exiting the program completely.

Which of these happens - dealing with it or aborting the program – depends on whether you have provided a **rescue** clause. If you haven't provided such a clause, the program terminates; if you have, control flows to the rescue clause.

Raising an Exception

Ruby has some predefined classes – **Exception** and its children – that help you to handle errors that can occur in your program. The following figure shows the Ruby exception hierarchy.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Figure 8.1. Ruby exception hierarchy

```
Exception
    ├─ fatal  (used internally by Ruby)
    ├─ NoMemoryError
    ├─ ScriptError
    │   ├─ LoadError
    │   ├─ NotImplementedError
    │   └─ SyntaxError
    ├─ SignalException
    │   └─ Interrupt
    ├─ StandardError
    │   ├─ ArgumentError
    │   ├─ IOError
    │   │   └─ EOFError
    │   ├─ IndexError
    │   ├─ LocalJumpError
    │   ├─ NameError
    │   │   └─ NoMethodError
    │   ├─ RangeError
    │   │   └─ FloatDomainError
    │   ├─ RegexpError
    │   ├─ RuntimeError
    │   ├─ SecurityError
    │   ├─ SystemCallError
    │   │   └─ system-dependent exceptions (Errno::xxx)
    │   ├─ ThreadError
    │   ├─ TypeError
    │   └─ ZeroDivisionError
    ├─ SystemExit
    └─ SystemStackError
```

Reference: The above figure is from the Programming Ruby book.

The following method raises an exception whenever it's called. It's second message will never be printed. Program p043raise.rb

```
def raise_exception
  puts 'I am before the raise.'
  raise 'An error has occured'
  puts 'I am after the raise'
end
raise_exception
```

The output is:

```
>ruby p043raise.rb
I am before the raise.
raise.rb:3:in `raise_exception': An error has occured
(RuntimeError)
      from raise.rb:6
>Exit code: 1
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The **raise** method is from the **Kernel** module. By default, **raise** creates an exception of the **RuntimeError** class. To raise an exception of a specific class, you can pass in the class name as an argument to **raise**. Refer program p044inverse.rb

```
def inverse(x)
  raise ArgumentError, 'Argument is not numeric' unless x.is_a?
Numeric
  1.0 / x
end
puts inverse(2)
puts inverse('not a number')
```

The output is:

```
>ruby p044inverse.rb
0.5
inverse.rb:2:in `inverse': Argument is not numeric (ArgumentError)
      from inverse.rb:6
>Exit code: 1
```

Remember, methods that act as queries are often named with a trailing **?**. **is_a?** is a method in the **Object** class and returns true or false. The

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

unless modifier when tacked at the end of a normal statement means execute the preceding expression unless condition is true.

To be even more specific about an error, you can define your own Exception subclass:

```
class NotInvertibleError < StandardError
end
```

Handling an Exception

To do exception handling, we enclose the code that could raise an exception in a **begin/end** block and use one or more **rescue** clauses to tell Ruby the types of exceptions we want to handle.

The program p045handexcp.rb illustrates this.

```
def raise_and_rescue
  begin
    puts 'I am before the raise.'
    raise 'An error has occured.'
    puts 'Iam after the raise.'
  rescue
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
    puts 'I am rescued.'
  end
  puts 'I am after the begin block.'
end
raise_and_rescue
```

The output is:

```
>ruby p045handexcp.rb
I am before the raise.
I am rescued.
I am after the begin block.
>Exit code: 0
```

Observe that the code interrupted by the exception never gets run. Once the exception is handled, execution continues immediately after the **begin** block that spawned it.

If you write a **rescue** clause with no parameter list, the parameter defaults to **StandardError**. Each **rescue** clause can specify multiple exceptions to catch. At the end of each **rescue** clause you can give Ruby the name of a local variable to receive the matched exception. The parameters to the **rescue** clause can also be arbitrary expressions

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

(including method calls) that return an **Exception** class. If we use **raise** with no parameters, it re-raises the exception.

You can stack **rescue** clauses in a **begin/rescue** block. Exceptions not handled by one **rescue** clause will trickle down to the next:

```
begin
  # …
rescue OneTypeOfException
  # …
rescue AnotherTypeOfException
  # …
end
```

For each **rescue** clause in the begin block, Ruby compares the raised Exception against each of the parameters in turn. The match will succeed if the exception named in the **rescue** clause is the same as the type of the currently thrown exception, or is a superclass of that exception.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

If you want to interrogate a rescued exception, you can map the **Exception** object to a variable within the **rescue** clause, as shown in the program p046excpvar.rb

```
begin
  raise 'A test exception.'
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
end
```

The output is:

```
>ruby p046excpvar.rb
A test exception.
["excpvar.rb:2"]
>Exit code: 0
```

If you need the guarantee that some processing is done at the end of a block of code, regardless of whether an exception was raised then the **ensure** clause can be used. **ensure** goes after the last **rescue** clause and contains a chunk of code that will always be executed as the block terminates. The **ensure** block will always run.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Some common exceptions are shown in the following table (courtesy the book Ruby For Rails):

Exception name	Common reason(s)	How to raise it
`RuntimeError`	This is the default exception raised by the `raise` method.	`raise`
`NoMethodError`	An object is sent a message it can't resolve to a method name.	`a = Object.new` `a.some_unknown_method_name`
`NameError`	The interpreter hits an identifier it can't resolve as a variable or method name.	`a = some_random_identifier`
`IOError`	This is caused by reading a closed stream, writing to a read-only stream, and similar operations.	`STDIN.puts("Don't write to STDIN!")`
`Errno::`*`error`*	This family of errors relates to file IO.	`File.open(-12)`
`TypeError`	A method receives an argument it can't handle.	`a = 3 + "can't add a string to a number!"`
`ArgumentError`	This is caused by using the wrong number of arguments.	`def m(x); end; m(1,2,3,4,5)`

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Assignment: Previously you had written a program that swapped the contents of two text files. Modify that program to include exception handling.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Access Control

The only easy way to change an object's state in Ruby is by calling one of its methods. Control access to the methods, and you have controlled access to the object. A good rule of the thumb is never to expose methods that could leave an object in an invalid state.

Ruby gives you three levels of protection.

- **Public methods** can be called by everyone – no access control is enforced. A class's instance methods (these do not belong only to one object; instead, every instance of the class can call them) are public by default; anyone can call them. The **initialize** method is always private.
- **Protected methods** can be invoked only by objects of the defining class and its subclasses. Access is kept within the family.
- **Private methods** cannot be called with an explicit receiver – the receiver is always **self**. This means that private methods can be called only in the context of the current object; you cannot invoke another object's private methods.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Access control is determined dynamically, as the program runs, not statically. You will get an access violation only when the code attempts to execute the restricted method.

Let's refer to the program p047classaccess.rb below.

```
class ClassAccess
  def m1                # this method is public
  end
  protected             # subsequent methods will be 'protected'
  def m2                # this method is protected
  end
  def m3                # this method is protected
  end
  private               # subsequent methods will be 'private'
  def m4                # this method is private
  end
  def m5                # this method is private
  end
end
```

Alternatively, you can set access levels of named methods by listing them as arguments to the access control functions.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
class ClassAccess
  def m1                # this method is public
  end
# ... and so on
public :m1
protected :m2, :m3
private :m4, :m5
end
```

Peter Cooper the author of Beginning Ruby has this example (p047zclassaccess.rb) for 'protected' access control:

```
class Person
  def initialize(age)
    @age = age
  end
  def age
    @age
  end
  def age_difference_with(other_person)
    (self.age - other_person.age).abs
  end
  protected :age
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
end
fred = Person.new(34)
chris = Person.new(25)
puts chris.age_difference_with(fred)
puts chris.age
```

The output is:

```
>ruby p047zclassaccess.rb
9
p047zclassaccess.rb:16: protected method `age' called for
#<Person:0x2bae148 @age=25> (NoMethodError)
>Exit code: 1
```

The preceding example uses a protected method so that the age method cannot be used directly, except within any method belonging to an object of the Person class. However, if age were made private, the preceding example would fail because other_person.age would be invalid. That's because private makes methods accessible only by methods of a specific object.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Note that when you use age directly, on the last line, Ruby throws an exception.

Instance variables are not directly accessible outside the class.

Top-level methods

When you write code at the top level, Ruby provides you automatically with a default **self**. This object is a direct instance of **Object**. When you ask it to describe itself

puts self

it says:

main

The object **main** is the current object as soon as your program starts up.

Suppose you define a method at the top level:

```
def talk
  puts "Hello"
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Who, or what, does the method belong to? It's not inside a class or module definition block, so it doesn't appear to be an instance method of a class or module. It's not attached to any particular object (as in def obj.talk). What is it? Top-level methods are *private instance methods of the* **Kernel** *module.*

Because top-level methods are private, you can't call them with an explicit receiver; you can only call them by using the implied receiver, **self**. That means **self** must be an object on whose method search path the given top-level method lies. But *every* object's search path includes the **Kernel** module, because the class **Object** mixes in **Kernel**, and every object's class has **Object** as an ancestor. That means you can *always* call any top-level method, wherever you are in your program. It also means you can *never* use an explicit receiver on a top-level method.

From our earliest examples onward, we've been making bareword-style calls to **puts** and **print**, like this one:

```
puts "Hello"
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

puts *and* **print** *are built-in private instance methods of* **Kernel**. That's why you can - indeed, must - call them without a receiver.

Accessor methods

Encapsulation is achieved when the instance variables are private to an object and you have public getters and setters (in Ruby, we call them attribute readers and attribute writers). To make instance variables available, Ruby provides accessor methods that return their values. The program p048accessor.rb illustrates the same.

```
# p048accessor.rb
# First without accessor methods
class Song
  def initialize(name, artist)
    @name     = name
    @artist   = artist
  end
  def name
    @name
  end
  def artist
    @artist
  end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
end

song = Song.new("Brazil", "Ricky Martin")
puts song.name
puts song.artist

# Now, with accessor methods
class Song
  def initialize(name, artist)
    @name     = name
    @artist   = artist
  end
  attr_reader :name, :artist  # create reader only
  # For creating reader and writer methods
  # attr_accessor :name
  # For creating writer methods
  # attr_writer :name
end

song = Song.new("Brazil", "Ricky Martin")
puts song.name
puts song.artist
```

Are instance variables inherited by a sub-class?

David Black the author of Ruby for Rails has this to say: Instance variables are per-object, not per-class, and they're not inherited. But if

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

a method uses one, and that method is available to subclasses, then it will still use the variable -- but "the variable" in the sense of one per object. See the following program - p049instvarinherit.rb:

```
class C
  def initialize
    @n = 100
  end

  def increase_n
    @n *= 20
  end
end

class D < C
  def show_n
    puts "n is #{@n}"
  end
end

d = D.new
d.increase_n
d.show_n
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The output is:

```
>ruby p049instvarinherit.rb
n is 2000
>Exit code: 0
```

The @n in D's methods is the same (for each instance) as the one in C.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Syntactic sugar

Programmers use the term *syntactic sugar* to refer to special rules that let you write your code in a way that doesn't correspond to the normal rules but that is easier to remember how to do and looks better.

Let's say we want to set the name of a dog. As a starting point, name can be set along with everything else at object creation time, as in example – p050newdog.rb

```
class NewDog
  def initialize(breed, name)
    @breed = breed
    @name = name
  end
  attr_reader :breed, :name   # create reader only
end

nd = NewDog.new('Doberman', 'Benzy')
puts nd.name
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Let's write a set_name method that allows us to set, or reset, the name of an existing dog. We'll also rewrite the **initialize** method so that it doesn't expect a name:

```
class NewDog
  def initialize(breed)
    @breed = breed
  end
  attr_reader :breed, :name   # create reader only

  # setter method
  def set_name(nm)
    @name = nm
  end
end

nd = NewDog.new('Doberman')
nd.set_name('Benzy')
puts nd.name
```

Ruby allows you to define methods that end with an equal sign (=). Let's replace set_name with a method called name=

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
def name=(nm)
  @name = nm
end
```

name= does exactly what set_name did, and in spite of the slightly odd method name, you can call it just like any other method:

```
nd.name=('Benzy')
```

Here's the modified example –

```
class NewDog
  def initialize(breed)
    @breed = breed
  end
  attr_reader :breed, :name   # create reader only

  # setter method
  def name=(nm)
    @name = nm
  end
end

nd = NewDog.new('Doberman')
nd.name=('Benzy')
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
puts nd.name
```

The equal sign gives you that familiar "assigning a value to something" feeling, so you know you're dealing with a setter method. It still looks odd, but Ruby takes care of that, too.

Ruby gives you some *syntactic sugar* for calling setter methods. Instead of this

```
nd.name=('Benzy')
```

you're allowed to do this:

```
nd.name = 'Benzy'
```

When the interpreter sees the message "name" followed by " =", it automatically ignores the space before equal sign and reads the single message "name=" - a call to the method whose name is name=, which we've defined. As for the right-hand side: parentheses are optional on single arguments to methods, so you can just put 'Benzy' there and it will be picked up as the argument to the name= method.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Method calls using the equal-sign syntax are common in Rails applications.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Mutable and Immutable Objects

Mutable objects are objects whose state can change. Immutable objects are objects whose state never changes after creation.

Immutable objects have many desirable properties:

- Immutable objects are thread-safe. Threads cannot corrupt what they cannot change.
- Immutable objects make it easier to implement encapsulation. If part of an object's state is stored in an immutable object, then accessor methods can return that object to outside callers, without fear that those callers can change the object's state.
- Immutable objects make good hash keys, since their hash codes cannot change.

In Ruby, Mutability is a property of an instance, not of an entire class. Any instance can become immutable by calling **freeze**.

Freezing Objects

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The **freeze** method in class **Object** prevents you from changing an object, effectively turning an object into a constant. After we freeze an object, an attempt to modify it results in **TypeError**. The following program (p050xfreeze.rb) illustrates this:

```
str = 'A simple string. '
str.freeze
begin
  str << 'An attempt to modify.'
rescue => err
  puts "#{err.class} #{err}"
end
# The output is - TypeError can't modify frozen string
```

However, **freeze** operates on an object reference, not on a variable. This means that any operation resulting in a new object will work. This is illustrated by the following example:

```
str = 'Original string - '
str.freeze
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
str += 'attachment'
puts str
# Output is - Original string - attachment
```

The expression str + 'attachment' is evaluated to a new object, which is then assigned to str. The object is not changed, but the variable str now refers to a new object. A method **frozen?** tells you whether an object is frozen or not.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Object Serialization

Java features the ability to *serialize* objects, letting you store them somewhere and reconstitute them when needed. Ruby calls this kind of serialization *marshaling*.

We will write a basic class p051gamecharacters.rb just for testing *marshalling*.

```
class GameCharacter
  def initialize(power, type, weapons)
    @power = power
    @type = type
    @weapons = weapons
  end
    attr_reader :power, :type, :weapons
end
```

The program p052dumpgc.rb creates an object of the above class and then uses **Marshal.dump** to save a serialized version of it to the disk.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
require 'p051gamecharacters.rb'
gc = GameCharacter.new(120, 'Magician', ['spells',
'invisibility'])
puts gc.power.to_s + ' ' + gc.type + ' '
gc.weapons.each do |w|
  puts w + ' '
end

File.open('game', 'w+') do |f|
  Marshal.dump(gc, f)
end
```

The program p053loadgc.rb uses **Marshal.load** to read it in.

```
require 'p051gamecharacters.rb'
File.open('game') do |f|
      @gc = Marshal.load(f)
end

puts @gc.power.to_s + ' ' + @gc.type + ' '
@gc.weapons.each do |w|
  puts w + ' '
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Constants

A Ruby constant is a reference to an object. Constants are created when they are first assigned to (as per the rules below). In the current implementation of Ruby, *reassignment of a constant* generates a warning but not an error as shown in this trivial example – p054constwarn.rb

```
# p054constwarn.rb
A_CONST = 10
A_CONST = 20
```

Produces a warning:

```
p054constwarn.rb:3: warning: already initialized constant A_CONST
```

Although constants should not be changed, you can *modify* the internal states of the objects they reference, as seen in p055constalter.rb

```
# p055constalter.rb
A_CONST = "Doshi"
B_CONST = A_CONST
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
A_CONST[0] = "J" # alter string referenced by constant
puts A_CONST # displays Joshi
puts B_CONST # also displays Joshi
```

You can find examples of this kind of operation (*modify*) in the Rails source code, where constants figure prominently and the objects they represent undergo fairly frequent changes.

- Constants defined within a class or module may be accessed anywhere within the class or module.
- Outside the class or module, they may be accessed using the scope operator, **::** prefixed by an expression that returns the appropriate class or module.
- Constants defined outside any class or module may be accessed as it is or by using the scope operator with no prefix.
- Constants may not be defined in methods.
- Constants may be added to existing classes and modules from the outside by using the class or module name and the scope operator before the constant name. The program p056const.rb shows all of this.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
# p056const.rb
OUTER_CONST = 99
class Const
  def get_const
    CONST
  end
  CONST = OUTER_CONST + 1
end
puts Const.new.get_const
puts Const::CONST
puts ::OUTER_CONST
puts Const::NEW_CONST = 123
```

Another elaborate example on own methods in a class is p057mymethods2.rb

Here we shall also see how to define class methods.

```
# variables and methods start lowercase
$glob = 5              # global variables start with $
class TestVar          # class name constant, start uppercase
 @@cla = 6             # class variables start with @@
 CONST_VAL = 7         # constant style, all caps, underscore
 def initialize(x)
   @inst = x           # instance variables start with @
   @@cla += 1          # each object shares @@cla
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
 end
 def self.cla          # class method, getter
   @@cla
 end
 def self.cla=(y)      # class method, setter, also TestVar.
  @@cla = y
 end
 def inst              # instance method, getter
  @inst
 end
 def inst=(i)          # instance method, setter
  @inst = i
 end
end
puts $glob
test = TestVar.new(3)
puts TestVar.cla       # calls getter
puts test.inspect      # gives object ID and instance vars
TestVar.cla = 4        # calls setter
test.inst=8            # calls setter
puts TestVar.cla
puts test.inst         # calls getter
other = TestVar.new(17)
puts other.inspect
puts TestVar.cla
```

Some questions asked by the members

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Can we have an anonymous class in Ruby?
Yes. Refer page 382 of the second edition of the book Programming Ruby, by Dave Thomas

Does Ruby have Multiple Inheritance?
Ruby uses modules to implement mix-ins that simulate multiple inheritance.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Modules/Mixins

Modules are similar to classes in that they hold a collection of methods, constants, and other module and class definitions. Unlike classes, you cannot create objects based on modules; instead, you specify that you want the functionality of a particular module to be added to the functionality of a class, or of a specific object.

Modules serve two purposes:

- First they act as namespace, letting you define methods whose names will not clash with those defined elsewhere. The examples p058mytrig.rb, p059mymoral.rb and p060usemodule.rb illustrates this.

```
# p058mytrig.rb
module Trig
  PI = 3.1416
  # class methods
  def Trig.sin(x)
    # ...
  end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
  def Trig.cos(x)
    # ...
  end
end

# p059mymoral.rb
module Moral
  VERY_BAD = 0
  BAD      = 1
  def Moral.sin(badness)
    # ...
  end
end

# p060usemodule.rb
require 'p058mytrig'
require 'p059mymoral'
y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::VERY_BAD)
```

- Second, they allow you to share functionality between classes – if a class mixes in a module, that module's instance methods become available as if they had been defined in the class. They get mixed in. The program p061mixins.rb illustrates this:

```
# p061mixins.rb
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
module D
  def initialize(name)
    @name =name
  end
  def to_s
    @name
  end
end

module Debug
  include D
  # Methods that act as queries are often
  # named with a trailing ?
  def who_am_i?
    "#{self.class.name} (\##{self.object_id}): #{self.to_s}"
  end
end

class Phonograph
  # the include statement simply makes a reference
  # to a named module
  # If that module is in a separate file, use require
  # to drag the file in
  # before using include
  include Debug
    # ...
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
class EightTrack
  include Debug
  # ...
end

ph = Phonograph.new("West End Blues")
et = EightTrack.new("Real Pillow")
puts ph.who_am_i?
puts et.who_am_i?
```

SYNTAX OF **require/load** VS. SYNTAX OF **include** - You may have noticed that when you use **require** or **load**, you put the name of the item you're requiring or loading in quotation marks, but with **include**, you don't. **require** and **load** take strings as their arguments, whereas **include** takes the name of a module, in the form of a constant. The requirements to **require** and **load** are usually literal strings (in quotation marks), but a string in a variable will also work.

Some more examples –

p062stuff.rb

```
# A module may contain constants, methods and classes.
# No instances
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
module Stuff
  C = 10
  def Stuff.m(x)  # prefix with the module name for a class method
    C*x
  end
  def p(x)        # an instance method, mixin for other classes
    C + x
  end
  class T
    @t = 2
  end
end
puts Stuff::C     # Stuff namespace
puts Stuff.m(3)   # like a class method
x = Stuff::T.new
# uninitialized constant error, if you try the following
# puts C
```

p063stuffusage.rb

```
require 'p062stuff' # loads Stuff module from p062stuff.rb
                    # $: is a system variable -- contains the path
                    # for loads
class D
  include Stuff     # refers to the loaded module
  puts Stuff.m(4)
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
d = D.new
puts d.p(5)          # method p from Stuff
puts $:              # array of folders to search for load
$: << "c:/"          # add a folder to the load path
puts $:
puts Stuff.m(5)      # Stuff class methods not called from D object
```

The main difference between inheriting from a class and mixing in a module is that you can mix in more than one module. No class can inherit from more than one class. In cases where you want numerous extra behaviors for a class's instances—and you don't want to stash them all into the class's superclass—you can use modules to organize your code in a more granular way. Each module can add something different to the methods available through the class. Class names tend to be nouns, while module names are often adjectives (like Stack versus Stacklike).

The Rails source code makes heavy use of modules, in particular the technique of *reopening* the definition bodies of both classes and modules.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Self - The current/default object

At every point when your program is running, there is one and only one **self**- the current or default object accessible to you in your program. You can tell which object **self** represents by following a small set of rules.

Top level context

The top level context is before you have entered any other context, such as a class definition. Therefore the term top level refers to program code written outside of a class or module. If you open a new text file and type:

```
x = 1
```

you have created a top level local variable x. If you type

```
def m
end
```

you have created a top level method - a method that is not defined as an instance method of a particular class or module.

Ruby provides you with a start-up **self** at the top level. If you type: **puts self** it displays **main** - a special term the default **self** object uses to refer to itself.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Self inside class and module definitions

In a class or module definition, **self** is the class or module object.

```
# p063xself1.rb
class S
  puts 'Just started class S'
  puts self
  module M
    puts 'Nested module S::M'
    puts self
  end
  puts 'Back in the outer level of S'
  puts self
end
```

The output is:

```
>ruby p063xself1.rb
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
Just started class S
S
Nested module S::M
S::M
Back in the outer level of S
S
>Exit code: 0
```

Self in instance method definitions

At the time the method definition is executed, the most you can say is that self inside this method will be some future object that has access to this method.

```
# p063xself2.rb
class S
  def m
    puts 'Class S method m:'
    puts self
  end
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
s = S.new
s.m
```

The output is:

```
>ruby p063xself2.rb
Class S method m:
#<S:0x2835908>
>Exit code: 0
```

The output **#<S:0x2835908>** is Ruby's way of saying "an instance of S".

Self in singleton-method and class-method definitions

Singleton methods - those attached to a particular object can be called by only one object. When a singleton method is executed, **self** is the object that owns the method, as shown below:

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
# p063xself3.rb
obj = Object.new
def obj.show
  print 'I am an object; '
  puts "here's self inside a singleton method of mine:"
  puts self
end
obj.show
print 'And inspecting obj from outside, '
puts "to be sure it's the same object:"
puts obj
```

The output of the above example is:

```
>ruby p063xself3.rb
I am in object: here's self inside a singleton method of mine:
#<Object:0x2835688>
And inspecting obj from outside, to be sure it's the same object:
#<Object:0x2835688>
>Exit code: 0
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Class methods are defined as singleton methods for class objects. Refer the following program:

```
# p063xself4.rb
class S
  def S.x
    puts "Class method of class S"
    puts self
  end
end
S.x
```

The output is:

```
>ruby p063xself4.rb
Class method of class S
S
>Exit code: 0
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

self inside a singleton method (a class method, in this case) is the object whose singleton method it is.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Regular Expressions

Regular expressions, though cryptic, is a powerful tool for working with text. Ruby has this feature built-in. It's used for pattern-matching and text processing.

Many people find regular expressions difficult to use, difficult to read, un-maintainable, and ultimately counterproductive. You may end up using only a modest number of regular expressions in your Rails applications. Becoming a regular expression wizard isn't a prerequisite for Rails programming. However, it's advisable to learn at least the basics of how regular expressions work.

A regular expression is simply a way of specifying a *pattern* of characters to be matched in a string. In Ruby, you typically create a regular expression by writing a pattern between slash characters (/*pattern*/). In Ruby, regular expressions are objects (of type **Regexp**) and can be manipulated as such. // is a regular expression and an instance of the **Regexp** class, as shown below.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
//.class    # Regexp
```

You could write a pattern that matches a string containing the text *Pune* or the text *Ruby* using the following regular expression:
`/Pune|Ruby/`

The forward slashes delimit the pattern, which consists of the two things we are matching, separated by a pipe character (|). The pipe character means "either the thing on the right or the thing on the left," in this case *Pune* or *Ruby*.

The simplest way to find out whether there's a match between a pattern and a string is with the **match** method. You can do this in either direction: Regular expression objects and string objects both respond to **match**. If there's no match, you get back **nil**. If there's a match, it returns an instance of the class **MatchData**. We can also use the match operator =~ to match a string against a regular expression. If the pattern is found in the string, **=~** returns its starting position, otherwise it returns **nil**.

```
/Ruby/.match("The future is Ruby")
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
# it returns <MatchData:0x2c9b024>
"The future is Ruby" =~ /Ruby/
# it returns 14
```

The possible components of a regular expression include the following:

Literal characters

Any literal character you put in a regular expression matches *itself* in the string.

```
/a/
```

This regular expression matches the string "a", as well as any string containing the letter "a".

Some characters have special meanings to the regexp parser. When you want to match one of these special characters *as itself*, you have to *escape* it with a backslash (\). For example, to match the character ? (question mark), you have to write this:

```
/\?/
```

The backslash means "don't treat the next character as special; treat it as itself."

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The special characters include ^, $, ? , ., /, \, [,], {, }, (,), +, and *.

The wildcard character . (dot)

Sometimes you'll want to match *any character* at some point in your pattern. You do this with the special wildcard character . (dot). A dot matches any character with the exception of a newline.

This regular expression
```
/.ejected/
```
matches both "dejected" and "rejected". It also matches "%ejected" and
"8ejected". The wildcard dot is handy, but sometimes it gives you more matches than you want. However, you can impose constraints on matches while still allowing for multiple possible strings, using *character classes*.

Character classes

A character class is an explicit list of characters, placed inside the regular expression in square brackets:

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
/[dr]ejected/
```

This means “match either *d* or *r*, followed by *ejected*. This new pattern matches either “dejected” or “rejected” but not “&ejected”. A character class is a kind of quasi-wildcard: It allows for multiple possible characters, but only a limited number of them.

Inside a character class, you can also insert a *range* of characters. A common case is this, for lowercase letters:

```
/[a-z]/
```

To match a hexadecimal digit, you might use several ranges inside a character class:

```
/[A-Fa-f0-9]/
```

This matches any character *a* through *f* (upper- or lowercase) or any digit.

Sometimes you need to match any character *except* those on a special list. You may, for example, be looking for the first character in a string that is *not* a valid hexadecimal digit.
You perform this kind of negative search by *negating* a character class. To do so, you put a caret (^) at the beginning of the class. Here's the

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

character class that matches any character except a valid hexadecimal digit:

```
/[^A-Fa-f0-9]/
```

Some character classes are so common that they have special abbreviations.

Special escape sequences for common character classes

To match *any digit*, you can do this:

```
/[0-9]/
```

But you can also accomplish the same thing more concisely with the special escape sequence \d:

```
/\d/
```

Two other useful escape sequences for predefined character classes are these:

```
\w matches any digit, alphabetical character, or underscore (_).
\s matches any whitespace character (space, tab, newline).
```

Each of these predefined character classes also has a negated form. You can match *any character that is not a digit* by doing this:

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
/\D/
```

Similarly, \W matches *any character other than an alphanumeric character or underscore*, and \S matches *any non-whitespace character.*

A successful match returns a **MatchData** object.

Every match operation either succeeds or fails. Let's start with the simpler case: failure. When you try to match a string to a pattern, and the string doesn't match, the result is always nil:

```
/a/.match("b")
# nil
```

This nil stands in for the *false* or *no* answer when you treat the match as a true/false test.

Unlike nil, the **MatchData** object returned by a successful match has a Boolean value of true, which makes it handy for simple match/no-match tests. Beyond this, however, it also stores information about the match, which you can pry out of them with the appropriate methods: where the match began (at what character in the string), how much of the string it covered, what was captured in the parenthetical groups, and so forth.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

To use the **MatchData** object, you must first save it. Consider an example where we want to pluck a phone number from a string and save the various parts of it (area code, exchange, number) in groupings. Example p064regexp.rb

```
string = "My phone number is (123) 555-1234."
phone_re = /\((\d{3})\)\s+(\d{3})-(\d{4})/
m = phone_re.match(string)
unless m
      puts "There was no match..."
      exit
end
print "The whole string we started with: "
puts m.string
print "The entire part of the string that matched: "
puts m[0]
puts "The three captures: "
3.times do |index|
      puts "Capture ##{index + 1}: #{m.captures[index]}"
end
puts "Here's another way to get at the first capture:"
print "Capture #1: "
puts m[1]
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

In this code, we use the **string** method of **MatchData** (**puts m.string**) to get the entire string on which the match operation was performed. To get the part of the string that matched our pattern, we address the **MatchData** object with square brackets, with an index of 0 (**puts m[0]**). We also use the **times** method (**3.times do |index|**) to iterate exactly three times through a code block and print out the submatches (the parenthetical captures) in succession. Inside that code block, a method called **captures** fishes out the substrings that matched the parenthesized parts of the pattern. Finally, we take another look at the first capture, this time through a different technique: indexing the **MatchData** object directly with square brackets and positive integers,
each integer corresponding to a capture.

Here's the output:

```
>ruby p064regexp.rb
The whole string we started with: My phone number is (123) 555-
1234.
The entire part of the string that matched: (123) 555-1234
The three captures:
Capture #1: 123
Capture #2: 555
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
Capture #3: 1234
Here's another way to get at the first capture:
Capture #1: 123
>Exit code: 0
```

Read the Ruby-centric regular expression tutorial here, for a more detailed coverage on regular expressions.

The above topic has been adapted from the Ruby for Rails book.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Unit Testing

Unit testing is a method of testing your code in chunks -- typically methods. You write individual tests, which you collect into *test cases*. You bundle these test cases into a *test suite*. Because the tests are small and run quickly, you can run them frequently to ensure that your code works correctly. Most *test runners* (an interface for running your tests) even allow you to select a subset of your tests to run instead of running every test every time.

Basically, unit testing helps you write code more quickly and with more confidence. You write smaller units of code at a time, reducing the number of bugs you introduce. You see the bugs immediately as a failing test and know exactly where to look for them in the code.

It might seem backwards, but coding test first really does mean that you write your tests before you start writing code. Doing so gives you some boundaries for your coding: You know what to write and when to stop writing code. Ruby comes with one preinstalled, Nathaniel Talbott's

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Test::Unit framework that actually does most of the heavy lifting to make this process work. It creates the test runner and collects the tests into cases and suites for you.

A rule of thumb: Test anything that's likely to fail. You want to write tests to ensure that your method does what it's supposed to do with normal input. You should check that invalid input will be handled correctly. Finally, test your boundary conditions -- the extreme edges of your expected input.

The steps to writing unit tests are:

- **require 'test/unit'** and set your test class to inherit from **Test::Unit::TestCase**
- Write methods prefixed with **test_**
- **assert** things you decide should be true
- Run your tests and fix the bugs until everything passes

Here's a bare-bones testing implementation – p065my_first_test.rb

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

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
class MyFirstTest < Test::Unit::TestCase
  def test_for_truth
    assert true
  end
end
```

As you can see above, your testing class (which is called a test suite) inherits from **Test::Unit::TestCase** and each of your tests is a method called test_that_thing_I_wanted_to_test. Within those tests, you use assertions to test whether conditions are correct in each situation.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Let's take a look at what kinds of assertions exist for you to test with. Nearly every assertion takes an optional message string, which is used to provide more information in the case of test failure.

The first group of assertions covers the values returned by the method under test:

- assert(boolean, [message])
- assert_equal(expected, actual, [message])
- assert_not_equal(expected, actual, [message])
- assert_in_delta(expected_float, actual_float, delta, [message])
- assert_match(pattern, string, [message])
- assert_no_match(regexp, string, [message])
- assert_same(expected, actual, [message])
- assert_not_same(expected, actual, [message])

The second group of assertions tests which type of object you're dealing with:

- assert_nil(object, [message])
- assert_not_nil(object, [message])
- assert_instance_of(klass, object, [message])
- assert_kind_of(klass, object, [message])

The third group of assertions covers an object's ability to respond to a method or operator:

Keeping in mind that coding test first really does mean that you write your tests before you start writing code, we now write a test suite p066testradius.rb for a simple class p067radius.rb that stores the radius of a circle.

```
# p066testradius.rb
require 'test/unit'
require 'p067radius'
class TestRadius < Test::Unit::TestCase
  def test_key
    robj = Radius.new('78')
    assert_equal('78', robj.key)
  end
end
```

Let's step through the example and see what needs to be done. Lines 1 and 2 pull in the libraries you need: Test::Unit (line 1) and radius (line 2). Lines 3-8 define the first TestCase, a class for testing Radius objects. In line 3, you're making the new class a subclass of Test::Unit::TestCase. Lines 4-7 define your first test, the test_key method of the TestRadius class.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The test itself consists of a bit of fixture code -- creating a radius object (robj) -- and an assertion (more on this later). This assertion just says, "The result of calling the key method on the Radius object is equal to the string 78."

Because you're writing your application test first, you've written these tests before you've written any code. The test suite actually forms a Ruby program on its own. When you try to run it for the first time, it will fail:

```
>ruby p066testradius.rb
c:/ruby/lib/ruby/site_ruby/1.8/rubygems/custom_require.rb:27:in
`gem_original_require': no such file to load -- radius (LoadError)
      from
c:/ruby/lib/ruby/site_ruby/1.8/rubygems/custom_require.rb:27:in
`require'
      from p066testradius.rb:2
>Exit code: 1
```

Fortunately, this problem is easy to fix. Simply create an empty p067radius.rb file so you can load it when you run your tests. Running the test again gives:

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
>ruby p066testradius.rb
Loaded suite p066testradius
Started
E
Finished in 0.0 seconds.

  1) Error:
test_key(TestRadius):
NameError: uninitialized constant TestRadius::Radius
    p066testradius.rb:5:in `test_key'

1 tests, 0 assertions, 0 failures, 1 errors
>Exit code: 1
```

Better, but still not quite what you want. This error message means that your test has no Radius class to use. So, define the Radius class as follows:

```
class Radius
end
```

Running your test suite again gives:

```
>ruby p066testradius.rb
Loaded suite p066testradius
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
Started
E
Finished in 0.0 seconds.

  1) Error:
test_key(TestRadius):
ArgumentError: wrong number of arguments (1 for 0)
    p066testradius.rb:5:in `initialize'
    p066testradius.rb:5:in `test_key'

1 tests, 0 assertions, 0 failures, 1 errors
>Exit code: 1
```

To fix these errors, you're going to have to write some actual code. When you stepped away from the test suite, it still generated errors because there wasn't a real initializer. You can fix that by writing an initializer for the object.

```
class Radius
  def initialize(key)
    @key = key
  end
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Lines 2 and 3 make up the initializer method. This method takes an argument (called key) and sets an instance variable (called @key) to its value. When you run the test suite again, you get:

```
>ruby p066testradius.rb
Loaded suite p066testradius
Started
E
Finished in 0.0 seconds.

  1) Error:
test_key(TestRadius):
NoMethodError: undefined method `key' for #<Radius:0x2cc6224
@key="78">
    p066testradius.rb:6:in `test_key'

1 tests, 0 assertions, 0 failures, 1 errors
>Exit code: 1
```

You're almost there. The next step is to create the getter method key. Ruby makes this process easy. Your whole getter is shown on line 2:

```
class Radius
  attr_reader :key
  def initialize(key)
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
    @key = key
  end
end
```

You've written just enough code to pass your first test. The Radius class can be initialized (with a key) and has a getter method that returns the key. Now when you run your test suite, you see:

```
>ruby p066testradius.rb
Loaded suite p066testradius
Started
.
Finished in 0.0 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
>Exit code: 0
```

Consider writing some additional tests to make sure your code doesn't fail. This method is simple, but what happens if you pass in a key that's not a **String** object? If it's a **Fixnum** object, you can just convert it to a **String** object. If it's anything else, your **initialize** should return **nil**. Here's what your new tests should look like:

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

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
require 'p067radius'
class TestRadius < Test::Unit::TestCase
  def test_key
    robj = Radius.new('78')
    assert_equal('78', robj.key)
    robj = Radius.new(78)
    assert_equal('78', robj.key)
    robj = Radius.new([78])
    assert_nil(robj.key)
  end
end
```

You know that this code will fail, but run it anyway. You should see the following output:

```
>ruby p066testradius.rb
Loaded suite p066testradius
Started
F
Finished in 0.016 seconds.

  1) Failure:
test_key(TestRadius) [p066testradius.rb:8]:
<"78"> expected but was
<78>.

1 tests, 2 assertions, 1 failures, 0 errors
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
>Exit code: 1
```

The report from your last test run points out some new information. First of all, you get to see what a failure looks like, instead of an error. This failure is significant because it means that your test and code work, but not correctly. You've got a bug. Second, you see that the report shows only two assertions, even though you wrote three. The report shows only two assertions because **Test::Unit** won't continue past the first failure. (That way, you can focus on getting your code right one test at a time.)

To fix the first failure, ensure that **Fixnum** objects are converted to **String** objects by changing your code as follows:

```
class Radius
  attr_reader :key
  def initialize(key)
    @key = key
    if @key.class == Fixnum then
      @key = @key.to_s
    end
  end
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Output generated by re-running your test suite indicates that you've fixed the first of two bugs:

```
>ruby p066testradius.rb
Loaded suite p066testradius
Started
F
Finished in 0.015 seconds.

  1) Failure:
test_key(TestRadius) [p066testradius.rb:10]:
<nil> expected but was
<[78]>.

1 tests, 3 assertions, 1 failures, 0 errors
>Exit code: 1
```

Now you can fix the next failure with the code by converting arrays to strings with the **to_s** method:

```
class Radius
  attr_reader :key
  def initialize(key)
    @key = key
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
    if @key.class == Fixnum then
      @key = @key.to_s
    end
    if @key.class != String then
      @key = nil
    end
  end
end
```

Now your code passes all your tests:

```
>ruby p066testradius.rb
Loaded suite p066testradius
Started
.
Finished in 0.0 seconds.

1 tests, 3 assertions, 0 failures, 0 errors
>Exit code: 0
```

Nearly every test you write for Radius uses the robj object. Instead of creating the object each time, you can use a **setup** method to do it for you. **Test::Unit** has two special methods: **setup** and **teardown**. If a test class has a **setup** method, it will be called before any of the assertion

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

methods. Conversely, any clean-up code that is required *after* each test method runs can be placed in a method named **teardown**.

Let's add a **setup** method to our above test suite.

```
def setup
  robj = Radius.new('78')
end
def test_key
      assert_equal('78', robj.key)
# additional functionality removed for brevity
end
```

The entire topic has been adapted from the articles on this subject by Pat Eyler, Kevin Clark and the Ruby Cookbook.

An interesting read is The Power of Tests.

Assignment:
Write a test suite for a simple class called Student. This class stores a first name, a last name, and an age: a person's full name is available as a computed value.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Usage of TCPServer and TCPSocket Classes for Date and Time

We shall develop sockets-based networking applications using the Ruby language. You need to have a basic familiarity with BSD-style sockets. Nowadays it is safe just to say "sockets". Previously one had to say "BSD sockets" because there were multiple socket APIs fighting for domination. Eventually, in the Unix/POSIX world the Berkely Software Distribution (BSD) style socket API won out. Most *Nix programmers that do networking use BSD style sockets.

If you are accustomed to programming network connections in C or C++, you will be pleasantly surprised at how easy it is to program them in the Ruby programming language.

Basic Networking

Let us talk a little bit about basic networking.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Our discussion of networking focuses on both sides of a *client-server relationship*. The *client* requests that some action be performed, and the *server* performs the action and responds to the client. A common implementation of the request-response model is between World Wide Web browsers and World Wide Web servers. When a user selects a Web site to browse through a browser (the client application), a request is sent to the appropriate Web server (the server application). The server normally responds to the client by sending an appropriate HTML Web page.

Port

A port is not a physical device, but an abstraction to facilitate communication between a server and a client.

Ports are described by a 16-bit integer value. Hence, a machine can have a maximum of 65535 port numbers (ranging from 0 to 65535). The port numbers are divided into three ranges: the Well Known Ports, the Registered Ports, and the Dynamic and/or Private Ports. The Well Known Ports are those from 0 through 1023 (for example, port no. 80 is for http, port no. 25 is for smtp and so on). The Registered Ports are those

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

from 1024 through 49151. The Dynamic and/or Private Ports are those from 49152 through 65535.

Internet Addresses

These are the numerical host addresses that consist of four bytes such as 132.163.4.102. The IP address 127.0.0.1 (localhost) is a special address, called the *local loopback address,* which denotes the local machine.

Sockets

A socket represents a single connection between two network applications. These two applications nominally run on different computers, but sockets can also be used for inter-process communication on a single computer. Applications can create multiple sockets for communicating with each other. Sockets are bidirectional, meaning that either side of the connection is capable of both sending and receiving data.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Socket classes

Ruby has a rich and diverse set of sockets classes. These classes range from the standard **Socket** class (which mimics the BSD Sockets API) to more refined classes that focus on a particular protocol or application type.

The class hierarchy of the base sockets classes is as shown below:

```
IO
└─ BasicSocket
    ├─ IPSocket
    │   ├─ TCPSocket
    │   │   ├─ SOCKSSocket
    │   │   └─ TCPServer
    │   └─ UDPSocket
    ├─ Socket
    └─ UNIXSocket
        └─ UNIXServer
```

The above figure is from the book **Programming Ruby**.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The **TCPSocket** class supports sockets for the connection-based, reliable Transmission Control Protocol. A helper class for the creation of TCP server sockets is also available in the **TCPServer** class. To create a stream socket and connect it to a server, the **TCPSocket** class can be used.

The Date Time Server and Client

Now let us build a Ruby based DateTime server and client that displays on the client computer the date and time at the server location, using the Ruby socket API.

Here's the code for the Date Time Server - p068dtserver.rb.

```
# Date Time Server - server side using thread
# usage: ruby p068dtserver.rb
require "socket"

dts = TCPServer.new('localhost', 20000)
loop do
  Thread.start(dts.accept) do |s|
    print(s, " is accepted\n")
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
    s.write(Time.now)
    print(s, " is gone\n")
    s.close
  end
end
```

Explanation of the p068dtserver.rb code:
You first load the socket library with the **require** command. The **TCPServer** class is a helper class for building TCP socket servers. The **TCPServer.new('localhost', 20000)** statement creates a new socket identified by localhost and port number. The **Thread.start** creates and runs a new thread to execute instructions given in *block*. Any arguments passed to **Thread.start** are passed into the block. The **dts.accept** method waits for a connection on *dts*, and returns a new **TCPSocket** object connected to the caller. The **Kernel.loop** iterator calls the associated block (**do..end**) forever (or at least until you break out of the loop). We use **s.write(Time.now)** to write the current date and time on the server to the socket. Finally, we write **s.close** to close a socket using the **close** method. This method is inherited from the **IO** class, but it's available for each of the socket types.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Since we are testing these programs on our local machine, open a new command window, go to the folder containing your p068dtserver.rb program and type **ruby p068dtserver.rb**. The program runs and waits for a client to connect at port 20000.

Here's the code for the Date Time Client - p069dtclient.rb

```
# Date Time Client
# usage: ruby p069dtclient.rb
require 'socket'

sock = TCPSocket.new('127.0.0.1', 20000)
str = sock.recv(100)
print str
sock.close
```

Explanation of p069dtclient.rb code:
You first load the socket library with the **require** command. The statement **sock = TCPSocket.new('127.0.0.1', 20000)** opens a TCP connection to localhost on port 20000. The statement **str = sock.recv(100)** receives up

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

to 100 bytes from **sock**. We display the date-time string received from the server and finally we close the socket.

Summary

This topic has explored the classes that make networking application development possible in Ruby. It discussed the basic classes for sockets programming (such as the **Socket** class) and also the classes that help to simplify sockets programming in Ruby, such as **TCPSocket** and **TCPServer**.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

A Small Project using SMTP class

To test our knowledge of what we have learnt in Ruby, we shall build a small SMTP Client.

For this project, I am referring to Request for Comments (RFC's) 1869, 2554, 2821, 2822 and 3207.

I'll build the SMTP client and the explanation follows. Let's call it p070rubysmtp.rb

```
# p070rubysmtp.rb
require 'net/smtp'
user_from = "satish@puneruby.com"
user_to = "ashish@puneruby.com"
the_email = "From: satish@puneruby.com\nSubject: Hello\n\nEmail by
Ruby.\n\n"
# handling exceptions
begin
  Net::SMTP.start('auth.smtp.1and1.co.uk', 25, '1and1.co.uk',
'satish@puneruby.com', 'password', :login) do |smtpclient|
    smtpclient.send_message(the_email, user_from, user_to)
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
  end
rescue Exception => e
  print "Exception occured: " + e
end
```

Here's some explanation of the code:

- The **Net::SMTP** library provides functionality to send internet mail via SMTP, the Simple Mail Transfer Protocol. This library does NOT provide functions to compose internet mails. You must create them by yourself
- The line **Net::SMTP.start** creates a new **Net::SMTP** object, opens a TCP connection and connects to the server. 'auth.smtp.1and1.co.uk' is the IP address of your SMTP server and the port used is 25. Since we have called with a block, the newly-opened **Net::SMTP** object is yielded to the block, and automatically closed when the block finishes. The third argument is the domain name which you are on (the host to send mail from). The next three arguments are the login id, password and form of authentication used (here :login). SMTP authentication schemes

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

are represented with symbols (:login, :plain, :cram_md5). Any given SMTP server may support any or all of these schemes.

- In the line smtpclient.send_message(the_email, user_from, user_to) the_email is the message to be sent. Single CR and LF found in the string, are converted to the CRLF pair. You cannot send a binary message with this method. the_email should include both the message headers and body. user_from is a String representing the source mail address. user_to is a **String** or Array of Strings, representing the destination mail address or addresses
- The way we have used **Net::SMTP.start** it finishes the SMTP session and closes TCP connection
- We have also used **begin** and **rescue** for Exception handling

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Web Services and Distributed Programming

The basic point of distributed programming is to let computers communicate between themselves. The simplest and most common form of distributed programming is the web service. Web services work on top of HTTP: they generally involve sending an HTTP request to a certain URL (possibly including an XML document), and getting a response in the form of another XML document. Rather than showing this document to an end user the way a web browser would, the web service parses the XML response document and does something with it. Web services work well when there's a server with some interesting data and many clients who want it.

From a high-level view, web service implementations can be broken down into two categories: servers and clients. Most web services are based on one of three architectures: Representational State Transfer (REST), Simple Object Access Protocol (SOAP), or Extensible Markup Language Remote Procedural Calls (XML-RPC). REST is HTTP; XML-RPC

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

and SOAP are protocols that run on top of HTTP. XML-RPC isn't used much nowadays.

WSDL stands for Web Service Description Language. A WSDL file is an XML document that defines the interface to a SOAP service. WSDL files provide details about the methods that the service exposes, the methods' arguments and return values, and the encodings used for data travelling between the client and the server. Everything you would ever need to know about a SOAP service can be described in a WSDL file. WSDL files serve both as a form of documentation for SOAP services and as a key to automating many of the steps of building SOAP clients. Thus it's possible to read a WSDL file and find out everything you need to know about the API it describes. The primary use of WSDL is to automate the client code so that you don't have to explicitly write code to handle different data encodings, different mappings between object types, and so on. Most modern web service platforms automatically generate WSDL files and make them available for clients to download.

Writing a SOAP Client

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

The following example (p071soapclient.rb) is thanks to Kevin Marshall. Here we are going to call a remote method through a SOAP-based web service. The SOAP client sends an XML representation of a method call to a server, and gets back an XML representation of a return value. The whole process is complex, but Ruby's built-in SOAP library (implements both the client and server sides of the SOAP protocol, including support for WSDL) handles the low-level details for you, leaving you free to focus on using the results in your program. To write the SOAP service, we use Hiroshi Nakamura's **soap4r** library. It is part of the Ruby standard library and implements SOAP version 1.1. We use the SOAP RPC Driver in the **soap4r** library. The simple program prints a quote of the day. It uses the SOAP RPC Driver to connect to the SOAP web service at codingtheweb.com.

Details of this web service are available here:
http://www.strikeiron.com/WebServiceDetail.aspx?WSID=1061

The web service's WSDL file is available here:
http://webservices.codingtheweb.com/bin/qotd.wsdl

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

We can use the following site to decipher a WSDL file. Enter the location of the web services' WSDL file and it will report back the various methods/options available.
http://www.strikeiron.com/analyzer/onlineanalyzer.aspx

In our case, it tells us that the method to be used is *getQuote* and the service is *qotd*.

Here's our program p071soapclient.rb

```
require 'soap/rpc/driver'
driver = SOAP::RPC::Driver.new(
  'http://webservices.codingtheweb.com/bin/qotd',
      'urn:xmethods-qotd')
driver.add_method('getQuote')
puts driver.getQuote
```

Here, 'http://webservices.codingtheweb.com/bin/qotd' is the location of the web service (known as the endpoint URL). This is available in the tag <soap:address location> in the wsdl file. Also, 'urn:xmethods-qotd' is the namespace used by the service's documents.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Once the driver is set up, we define the web service method we want to call (getQuote). We can then call it like a normal Ruby method and display the result. Behind the scenes, the call to **add_method** actually defines a new method on the SOAP::RPC::Driver object. The SOAP library uses metaprogramming to create custom Ruby methods that act like SOAP methods.

Writing a PuneRuby SOAP server and client

Again thanks to Kevin Marshall for this too. Let's say that PuneRuby wants to host a SOAP-based web service (we are going to build a trivial one just to show how it is done) using a standalone server (that is, not as part of a Rails application).

Building our own SOAP server (program p072soapserver.rb) really only requires three simple steps:

1. Subclass the **SOAP::StandaloneServer** class. In the **initialize**, register the methods you want to expose and the arguments they should

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

take. Here we expose a method **sayhelloto** that expects one parameter, **username**:

```
require 'soap/rpc/standaloneServer'
class MyServer < SOAP::RPC::StandaloneServer
  def initialize(* args)
    super
    add_method(self, 'sayhelloto', 'username')
  end
end
```

2. Define the methods you exposed in Step 1:

```
class MyServer < SOAP::RPC::StandaloneServer
  def sayhelloto(username)
    "Hello, #{username}."
  end
end
```

3. Finally, set up and start your server. Our example runs on port 12321 on www.puneruby.com It's name is 'PuneRubyServer' and its namespace is 'urn:mySoapServer':

```
server = MyServer.new('PuneRubyServer','urn:mySoapServer',
                      'www.puneruby.com',12321)
trap('INT') {server.shutdown}
server.start
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

We trap interrupt signals so that we can stop our server from the command line.

We've now built a complete SOAP server. It uses the SOAP **StandaloneServer** and hosts one simple **sayhelloto** method that can be accessed at **http://217.160.200.122:12321/** with a namespace of "**urn:mySoapServer**".

To test your service, start your server in one Ruby session and then use the simple script (p073prclient.rb) below in another Ruby session to call the method it exposes:

```
require 'soap/rpc/driver'
driver =
  SOAP::RPC::Driver.new('http://217.160.200.122:12321/',
                        'urn:mySoapServer')
driver.add_method('sayhelloto', 'username')
puts driver.sayhelloto('PuneRuby')
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Ruby/Tk

Ruby/Tk provides Ruby with a graphical user interface (GUI). The Tk extension works on Windows, Mac OS X and Unix systems.

Previous versions of the Ruby One-Click Installer contained an (old) version of Tcl/Tk. Now this Ruby installer only contains the Ruby bindings to whatever version of Tcl/Tk you wish to install. It's recommended to use ActiveTcl. Download the file ActiveTcl8.4.13.0.261555-win32-ix86-threaded.exe and install it. Reboot your PC.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Tk works along a composition model – that is, you start by creating a container (such as a **TkFrame** or **TkRoot**) and then create widgets (GUI components) that populate it, such as buttons or labels. When you are ready to start the GUI, you invoke **Tk.mainloop**. The Tk engine then takes control of the program, displaying widgets and calling your code in response to GUI events.

Simple Tk applications

Let's understand the following program p074hellotk.rb

```
require 'tk'
hello = TkRoot.new {title "Hello World"}
Tk.mainloop
```

The Tk documentation shows the class hierarchy. Briefly, the hierarchy is as follows:

```
Object->TclTkIp->TkKernel->TkObject->TkWindow->TkRoot
Object->TclTkIp->TkKernel->TkObject->TkWindow->TkLabel->TkButton
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

There are many modules like:
Tk, Tk::Wm and many others.

TkKernel is the superclass of **TkObject**. The subclass redefines the class method **new** to take a block.
TkObject is the superclass of all widgets and has an included **Tk** module.
TkRoot class represents the root widget. The root widget is at the top of the Ruby/Tk widget hierarchy and has the included module **Tk::Wm** for communicating with a window manager. The methods introduced here are normally used as instance methods of **TkRoot**.

In the program, after loading the tk extension module, we create a root-level frame using **TkRoot.new**. With Tk you create widgets and then bind code blocks to them. When something happens (like the user clicking a widget), Tk runs the appropriate code block. In our program, we use the **title** and **minsize** instance methods (of module **Tk::Wm**) in the code block to **TkRoot.new**. We are now ready with our GUI and we invoke **Tk.mainloop**

Let us now add some widgets to the above program, namely a **TkLabel**. The modified program is p075hellotk1.rb

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
require 'tk'
hello = TkRoot.new do
  title "Hello World"
  # the min size of window
  minsize(400,400)
end
TkLabel.new(hello) do
  text 'Hello World'
  foreground 'red'
  pack { padx 15; pady 15; side 'left'}
end
Tk.mainloop
```

Here, we make a **TkLabel** widget (representing a label) as a child of the root frame, setting several options for the label. Finally, we pack the root frame and enter the main GUI event loop.

We also need to be able to get information back from our widgets while our program is running by setting up *callbacks* (routines invoked when certain events happen) and sharing data. The next example, p076hellotk2.rb does that.

```
require 'tk'
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
TkButton.new do
  text "EXIT"
  command { exit }
  pack('side'=>'left', 'padx'=>10, 'pady'=>10)
end
Tk.mainloop
```

Callbacks are very easy to setup. The **command** option takes **a Proc** object, which will be called when the callback fires. This means that the code block passed into the **command** method is run when the user clicks the button, allowing you to programmatically execute the same functionality that would be invoked on an actual button press. Note that the **Kernel#exit** method terminates your program here.

We shall now build a rudimentary GUI interface to our SOAP server hosted at www.puneruby.com. The program is the p077soapguiclient.rb

```
require 'soap/rpc/driver'
require 'tk'

class SOAPGuiClient
  def connect
    @buttonconnect.configure('text' => 'Reset')
    @buttonconnect.command { reset }
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
    begin
      driver =
SOAP::RPC::Driver.new('http://217.160.200.122:12321/',
                                      'urn:mySoapServer')
      driver.add_method('sayhelloto', 'username')
      s = driver.sayhelloto('Satish Talim')
    rescue Exception => e
      s = "Exception occured: " + e
    ensure
      @label.configure('text' => s)
    end
  end #connect

  def reset
    @label.configure('text' => "")
    @buttonconnect.configure('text' => 'Connect')
    @buttonconnect.command { connect }
  end # reset

#---
  def initialize
    root = TkRoot.new do
      title 'SOAP Client'
      # the min size of window
      minsize(535, 100)
    end
#---
    @label = TkLabel.new(root) do
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
      pack
    end
#---
    @buttonconnect = TkButton.new(root)
    @buttonconnect.configure('text' => 'Connect')
    @buttonconnect.command { connect }
    @buttonconnect.pack('side'=>'bottom')
#---
    Tk.mainloop
  end #initialize
end # class

SOAPGuiClient.new
#---
```

In the above program, we reconfigure a widget while it's running. Every widget supports the **configure** method, which takes a code block in the same manner as **new**. We have also modified the earlier p072soapserver.rb program, as below. We have included the **Logger** class.

```
require 'logger'
require 'soap/rpc/standaloneServer'
class MyServer < SOAP::RPC::StandaloneServer
  def initialize(* args)
    super
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
    add_method(self, 'sayhelloto', 'username')
    @log = Logger.new("soapserver.log", 5, 10*1024)
  end
  def sayhelloto(username)
    t = Time.now
    @log.info("#{username} logged on #{t}")
    "Hello, #{username} on #{t}."
  end
end

server =
MyServer.new('PuneRubyServer','urn:mySoapServer','www.puneruby.com
',12321)
trap('INT') {server.shutdown}
server.start
```

The **Logger** class helps write log messages to a file or stream. It supports time- or size-based rolling of log files. Messages can be assigned severities, and only those messages at or above the logger's current reporting level will be logged. Here, we log to a file called soapserver.log, which is rotated when it gets about 10K bytes and keeps up to five old files.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Using Ruby/MySQL

MySQL support in Ruby was made possible by Tomita Masahiro. He has developed a pure Ruby binding called Ruby/MySQL.

We need to install the same on our PC. You need an internet connection and do remember it does take some time to install. The installation is as shown below:

```
C:\>gem install mysql
Bulk updating Gem source index for: http://gems.rubyforge.org
Select which gem to install for your platform (i386-mswin32)
 1. mysql 2.7.1 (mswin32)
 2. mysql 2.7 (ruby)
 3. mysql 2.6 (ruby)
 4. mysql 2.5.1 (ruby)
 5. Cancel installation
> 1
Successfully installed mysql-2.7.1-mswin32
Installing ri documentation for mysql-2.7.1-mswin32...
Installing RDoc documentation for mysql-2.7.1-mswin32...

C:\>
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

This installs mysql-2.7.1-mswin32 driver which is faster and supports MySQL 4.1 and later. The documentation for this driver is here.

I will assume that you've already installed mySQL 4.1 or above on your PC and that you have it running and are familiar with the basics.

Now, run the mysql client program from the command line, as:

```
C:\>mysql
```

You should get the mysql prompt. Next, create a database ruby as:

```
mysql> create database ruby;
Query OK, 1 row affected (0.02 sec)
```

Next, create a table student in the database ruby as:

```
mysql> use ruby;
create table student (id VARCHAR(2), name VARCHAR(20), rank
VARCHAR(2));
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

As a first exercise we try to connect to the MySQL server and print all the names in the table student. Program p078rubymysql.rb

```
require 'mysql'

#my = Mysql.new(hostname, username, password, databasename)
con = Mysql.new('localhost', '', '', 'ruby')
rs = con.query('select * from student')
rs.each_hash {|h| puts h['name']}
con.close
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Ruby Tools

I have not gone into the details of some of the important tools that Ruby provides for programmers. Here's a brief idea:

irb

irb is an interactive interpreter – which means that instead of processing a file, it processes what you type during a session. irb is a great tool for testing Ruby code and for learning Ruby. Since I am using SciTE, I have not gone into the details of irb. You can refer to this and this resource for details of irb.

The debugger

Debugging – fixing errors – is part of programming. The Ruby debugging facility (found in the library file debug.rb) helps you debug a program by letting you run the program one instruction at a time, with pauses in between. During the pauses, you are presented with a prompt; at this

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

prompt, you can examine the values of variables, see where you are in a nested series of commands, and resume execution. You can also set *breakpoints* – places in your program where the debugger stops execution and presents you with a prompt. You can refer to this and this resource for details of the debugger.

Profiling

In programming terms, *profiling* means measuring how much use is made of system resources - time, principally - by different parts of your program. This starts to matter with longer programs, particularly programs that involve looping through instructions many times (for example, a program that reads in a long file and examines or modifies the contents of each line as it's read in). None of the examples on this site require profiling, because they're short and simple.

Profiling pinpoints the spots in a program that are using lots of system resources and therefore potentially slowing the program. The information provided by the profiler may lead you to tweak part of a program to make it run more efficiently; or, if there's no relatively easy

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

way around the resource bottleneck, it may lead you to rewrite part of the program in C, to make it run faster. You can refer to this resource for details of profiling Ruby code.

ri and RDoc

ri (Ruby Index) and **RDoc** (Ruby Documentation) are a closely related pair of tools for providing documentation about Ruby programs. ri is a command-line tool; the RDoc system includes the command-line tool rdoc. ri and rdoc are standalone programs; you run them from the command line.

RDoc is a documentation system. If you put comments in your program files (Ruby or C) in the prescribed RDoc format, rdoc scans your files, extracts the comments, organizes them intelligently (indexed according to what they comment on), and creates nicely formatted documentation from them. You can see RDoc markup in many of the C files in the Ruby source tree and many of the Ruby
files in the Ruby installation.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

ri dovetails with RDoc: It gives you a way to view the information that RDoc has extracted and organized. Specifically (although not exclusively, if you customize it), ri is configured to display the RDoc information from the Ruby source files. Thus on any system that has Ruby fully installed, you can get detailed information about Ruby with a simple command-line invocation of ri. Some more information is available here.

ERb

Ruby provides you with a program called ERb (Embedded Ruby), written by Seki Masatoshi. ERb allows you to put Ruby code inside an HTML file.

ERb reads a file—an ERb *document*—and prints it out again. You're allowed to insert Ruby programming instructions in the document (using a special syntax). When ERb hits the Ruby instructions, it executes them. Depending on what you've asked for, it either moves on or prints out the results of executing the instructions.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

ERb looms very large in the Ruby on Rails framework. Essentially, what you see on the screen when you connect to a Rails application is, in many cases, the output from an ERb document.

Some more information is available here.

All the details on this page has been adapted from David Black's very informative book – Ruby for Rails.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Java and Ruby

Similarities

As with Java, in Ruby…

- memory is managed for you via a garbage collector.
- there's public, private, and protected methods.
- you've got embedded doc tools (Ruby's is called RDoc). The docs generated by rdoc look very similar to those generated by javadoc. RDoc can produce fairly good content even if the source contains no comments.

Differences

Unlike Java, in Ruby…

- you don't need to compile your code. You just run it directly.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

- there's different GUI toolkits. Ruby users can try WxRuby, FXRuby, or the bundled-in Ruby Tk for example.
- you use the **end** keyword after defining things like classes, instead of having to put braces around blocks of code.
- you have **require** instead of import.
- all member variables are private. From the outside, you access everything via methods.
- parentheses in method calls are usually optional and often omitted.
- everything is an object, including numbers like 2 and 3.14159. Classes are objects! For example, Array is a constant name that is bound to the **Array** class object. To create a new object, we call **new** on the class object as in **a = Array.new**
- there are no primitives or data types
- variable names are just labels (not objects). They don't have a type associated with them.
- there's no type declarations. You just assign to new variable names as-needed and they just "spring up" (i.e. a = [1,2,3] rather than int[] a = {1,2,3};).
- it's **foo = Foo.new("hi")** instead of foo = new Foo("hi").

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

- the constructing and initializing phases of an object are separate and both can be over-ridden. The initialization is done via the **initialize** instance method while the construction is done via the **new** class method. **initialize** is not a constructor!
- you have "mixin's" instead of interfaces. mixins are examples of implementation inheritance, they are **not** equivalent to Java interfaces
- YAML tends to be favoured over XML.
- it's **nil** instead of null. Also, nil is a normal object; you can never get a null pointer error!
- there is no method overloading.
- it's much more common to put many classes in the same file.
- strings are mutable, unless you freeze them

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

JRuby Tutorial

Recently, JRuby has been gaining more and more attention in the Java and Ruby communities. Java is a powerful platform and there are millions of lines of Java code being written each month, that the world will have to live with for a long time from now. By leveraging Java the platform with the power of the Ruby programming language, programmers get the best from both worlds. You better not ignore JRuby any more!

What is JRuby?

JRuby is a 100% pure-Java implementation of the Ruby programming language that runs in the JVM. JRuby's creators, Thomas Enebo and Charles Nutter, have been hired by Sun to work on JRuby full time. The current JRuby release 0.9.9 is fully compatible with Ruby 1.8.5.

Ola Bini says that "JRuby is ready for prime time. Application developers should try their applications on JRuby NOW."

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

JRuby, as you will realize, is very easy to use. Here's a small **JRuby Tutorial** for you.

Download and Setup

- The JRuby distribution comes as a tar.gz file, namely jruby-bin-0.9.9.tar.gz
- Uncompress the archive; you should end up with a **jruby-0.9.9 folder**.
- In Windows, set the system environment variable **JRUBY_HOME** to C:\jruby-0.9.9 I am assuming that you have uncompressed JRuby to C:
- Also, set the system environment variable **path** to C:\jruby-0.9.9\bin;
- The JRuby distribution's bin directory contains the jruby.bat file that is used to run the JRuby interpreter. Run the command **jruby -version** from the command line to test that the JRuby is working.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

On my PC, it said:
ruby 1.8.5 (2007-04-23 rev 3539) [x86-jruby0.9.9]

Where to use JRuby?

a. *JRuby allows Ruby programs to use Java classes*. This is a powerful concept that JRuby now brings to Ruby users. My Ruby/Tk Tutorial has a program **p075hellotk1.rb** that uses Ruby/Tk a graphical user interface. However, the documentation for Ruby/Tk is extremely poor and I would be comfortable in using Java Swing instead. The code in JRuby for the program **p075hellotk1.rb** can be something like this:

```
# javaSwingHello.rb
require 'java' # Line 2
JFrame = javax.swing.JFrame
JLabel = javax.swing.JLabel
frame  = JFrame.new
jlabel = JLabel.new("Hello World")
frame.add(jlabel)
frame.setDefaultCloseOperation(JFrame::EXIT_ON_CLOSE)
frame.pack
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
frame.setVisible(true)
```

Run the above program from the command line as follows:
jruby javaSwingHello.rb

Note: Line 2: The second line of the above program enables JRuby's Java support and allows a Ruby program to use Java classes.

b. ***Calling JRuby from Java***. JRuby can just as easily be called from Java, using either the JSR 223 Scripting for Java 6 or the Apache Bean Scripting framework. More information on this is available in the JRuby Wiki

c. ***Running Rails with JRuby***. The advantages are obvious. To quote the JRuby Wiki

- Gives Rails the power and functionality of Java: Virtual Machine, application servers, and libraries.
- With future JVM and JRuby improvements, JRuby may be faster than Matz's Ruby Interpreter in running Rails

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

- Gets Rails in the door of Java shops by making Rails apps into Java-platform apps.

I personally feel that if you have a Java background and are now exploring Ruby, you should definitely explore JRuby further.

Resources:

- JRuby Home page
- JRuby Wiki
- Rails on JRuby
- JRuby Forum

Useful Articles:

- Internationalization in JRuby
- Connect JRuby to MySQL using JDBC

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Ruby Quirks

Ruby Quirks - peculiarity of behavior? I know this topic is debatable and remember 'one man's meat is another man's poison!

I plan to write down here (in no particular order), all the little Ruby quirks that I've picked up over time.

1. Peter Cooper, the author of the book 'Beginning Ruby' introduced me to Real-Time chat using an IRC client. On the #ruby channel at irc://irc.freenode.net/ I heard of this quirk:

```
class MotorCycle
  def initialize(make, color)
    @make, @color = make, color
  end
end

m = MotorCycle.new('Honda', 'blue')
m.instance_variable_set(:@make, 'Kawasaki')
m.instance_variable_set(:@gears, 4)
puts m.inspect
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Check the output of the above program. In the code above:
m.instance_variable_set(:@gears, 4)

sets the instance variable names by *symbol* to object, thereby frustrating the efforts of the class's author to attempt to provide proper encapsulation. The variable did not have to exist prior to this call.

Hal Fulton in his excellent book 'The Ruby Way' has this to say about instance_variable_set:

> It's true these methods are powerful and potentially dangerous. They should be used cautiously, not casually. But it's impossible to say whether encapsulation is violated without looking at *how these tools are used*. If they are used intentionally as part of a good design, then all is well. If they are used to violate the design, or to circumvent a bad design, then all is not well.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

2. In 2006, Shashank Date gave the PuneRuby members a presentation on 'Why Ruby Shines' and three points stood out - 'Expressions everywhere', 'Active Class Definitions' and 'Everything is an Object'.

Expressions everywhere - In Ruby, everything returns some value. Therefore a class definition is an expression and one can say something like:

```
c = class C
end
```

The value of c is nil.

Active Class Definitions - Look at the following program:

```
class C
  puts 'In class C'
end
```

When this class is read the first time, it executes puts and the output is - 'In class C'.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Everything is an Object - Classes in Ruby are first-class objects - each is an instance of class Class. When a new class is defined (typically using class Name ... end), an object of type Class is created and assigned to a constant (Name. in this case). Hal Fulton's suggests a *mantra* to be recited everyday - "Class *is an object*, and Object *is a class*."

3. If I want to swap two variables, I would normally use an additional temporary variable. In Ruby, this is not necessary:

```
x, y = y, x
```

will interchange the values of x and y.

4. Jaaron, a reader of the Learning Ruby Blog has this quirk for us. This one is well known and is the cause of much frustration.

```
x = 7
[1,2,3].each do |x|
end
```

If the name of a block parameter conflicts with the name of a local variable, the behavior is to assign the local variable to the argument. In

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

this case, the local variable x gets assigned the value 1, then the value 2, then the value 3. The value 7 is lost.

If you refer to Programming Ruby Second Edition eBook (page 99) it says:

> The while, until, and for loops are built into the language and do not introduce new scope; previously existing locals can be used in the loop, and any new locals created will be available afterward.
>
> The blocks used by iterators (such as loop and each) are a little different. Normally, the local variables created in these blocks are not accessible outside the block.
>
> However, if at the time the block executes a local variable already exists with the same name as that of a variable in the block, the existing local variable will be used in the block. Its value will therefore be available after the block finishes.
>
> The whole issue with variable scope and blocks is one that generates considerable discussion in the Ruby community. The current scheme

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

has definite problems (particularly when variables are unexpectedly aliased inside blocks), but at the same time no one has managed to come up with something that's both better and acceptable to the wider community. Matz is promising changes in Ruby 2.0.

5. Are instance variables inherited by a sub-class? David Black the author of Ruby for Rails has this to say: Instance variables are per-object, not per-class, and they're not inherited. But if a method uses one, and that method is available to subclasses, then it will still use the variable -- but "the variable" in the sense of one per object. See the following program:

```
class C
  def initialize
    @n = 100
  end

  def increase_n
    @n *= 20
  end
end

class D < C
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
  def show_n
    puts "n is #{@n}"
  end
end

d = D.new
d.increase_n
d.show_n
```

The output is:

```
>ruby instvarinherit.rb
n is 2000
>Exit code: 0
```

6. Morgan Schweers, a reader of the Learning Ruby Blog has this quirk for us. Imagine for a moment, that you want to be able to set a variable, but if it's not set, you default to a known value. You'd rather do it on a single line.

One of my co-workers tried this:

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
expand = defined?( expand ) ? expand : true
```

but 'expand' is *defined* by being on the left hand side, BEFORE the RHS is evaluated, so defined? returns true, but because expand hasn't got a value yet, it returns nil.

I tried:

```
expand = true unless defined?(expand)
```

and it doesn't help either, which really shocked me. I always believed that the postfix-conditional was evaluated before even beginning to evaluate the operation, but I was distinctly disabused of this notion by testing.

Note that 'expand?' operator returns nil if its argument (which can be an arbitrary expression) is not defined; otherwise it returns a description of that argument.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

I don't understand the reason for the behavior, and I think it's a bug, but I'd love to know a good language reason for it.

I am sure that you would have noticed many other Ruby quirks. I'd definitely like to hear and add them here.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Appendix A

Ruby Resources

Books/eBooks/Magazines

Ruby Code and Style Magazine
Free Ruby Books
O'Reilly's Ruby Books
The Little Book of Ruby
Learn to Program
Programming Ruby - First Edition
Whys Poignant Guide to Ruby
Ruby Cookbook
ObjectiveView
Ruby Tutorial

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Useful Links

About.com on Ruby
Ruby Manual
Ruby Documentation
The Ruby Language FAQ
Ruby Coding Convention
Ruby Quick Reference
Ruby Cookbook
More Cookbook
Ruby Weekly News
The Ruby FAQ
Ruby Language FAQ
Don Craig's Presentation
Ruby News
Ruby Notes
Ruby Jobs
Ruby Quiz
RubyIdioms
Ruby NewComers

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Misc. Ruby Programs
SciTE Text Editor
NotePad2 for Ruby
FreeRIDE Ruby IDE
Ruby jEdit
RubyGarden
Official Ruby Homepage
PuneRuby Homepage
Ruby Window's Installer
Rubyholic
Code Snippets
Java to Ruby
TryRuby
The Unofficial Ruby Usage Guide
Ruby/Tk FAQ
Ruby/Tk Tutorial
Ruby/Tk Documentation
Freelance Ruby Projects
Ruby MySQL Module
Ruby Reports

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Ruby Internals
Programming Language Comparison
Ruby Cheat Sheet
Ruby mySQL module
Test Driven Development
TIOBE Ruby Index
Ruby API Search

Blogs

PuneRuby blog
ChadFowler.com
Loud Thinking
Bruce Tate's Blog
PragDave
O'Reilly Ruby
Andy Hunt
Mindstorm

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Open Source Initiative

Forums/User Groups

Ruby-Talk Forum
SitePoint's Ruby Forum
Ruby IndiaGroup
TRUG

Companies in India working in Ruby/Rails

Better Labs, Pune
Codewalla, Pune
Chanakya Software Services, Pune
DancingBison, Pune
MangoSpring, Pune
Persistent Systems, Pune

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Thoughtworks, Pune
Allerin, Mumbai
Pinstorm, Mumbai
SurgeWorks, Mumbai
Urban Eye, Mumbai
V2Solutions, Mumbai
360 Degree Interactive, Chennai
Viamentis Technologies Pvt. Ltd., Chennai
BroadSpire, Chennai
Accenture Tech Labs, Bangalore
Aditi Technologies, Bangalore
Apptility, Bangalore
CircleSource, Bangalore
Itellix, Bangalore
Subex Azure, Bangalore
Satyam, Hyderabad
WarmlyYours, Hyderabad
Uzanto, New Delhi
Vinayak Solutions Pvt. Ltd., New Delhi
Induslogic, Noida

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Appendix B

Solutions to Assignments

1. Write a Ruby program that tells you how many minutes are there in a year (do not bother right now about leap years etc.).

```
puts 365*24*60
```

2. Write a Ruby program that asks for a numeric value of the temperature in degrees Fahrenheit. Finally, the program displays the equivalent value in degrees Centigrade. To format the output to say 2 decimal places, we can use the Kernel's **format** method. For example, if x = 45.5678 then format("%.2f", x) will return a string 45.57. Another way is to use the **round** function as follows **puts (x*100).round/100.0**
   Program p006ftoc.rb

```
# ftoc.rb
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
puts 'Enter temperature in Fahrenheit: '
STDOUT.flush
temp_in_fahrenheit = gets.chomp
temp_in_celsius = (((temp_in_fahrenheit.to_f - 32.0) / 9.0) *
5.0)
puts 'Temperature ' + temp_in_fahrenheit + ' degree Fahrenheit
= ' + format("%.2f", temp_in_celsius) + ' degree Celsius'
```

3. Write a Ruby program that asks for a year and then displays to the user whether the year entered by him/her is a *leap year* or not. Program p016leapyear.rb

```
=begin
Program to determine if a year is a leap year.
To determine if a year is a leap year, follow these steps:
1.   If the year is evenly divisible by 4, go to step 2.
Otherwise, go to step 5.
2.   If the year is evenly divisible by 100, go to step 3.
Otherwise, go to step 4.
3.   If the year is evenly divisible by 400, go to step 4.
Otherwise, go to step 5.
4.   The year is a leap year (it has 366 days).
5.   The year is not a leap year (it has 365 days).

The above logic is combined into a single if check below
=end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
# Get the input and determine if it is a leap year
puts "Enter the year: "
STDOUT.flush
input_year = gets.chomp.to_i
if ((input_year % 4 == 0) && (input_year % 100 > 0)) ||
(input_year % 400 == 0)
  puts "Year #{input_year} is a leap year"
else
  puts "Year #{input_year} is not a leap year"
end
```

4. Write a method **leap_year**. Accept a year value from the user, check whether it's a leap year and then display the number of minutes in that year. Program p017leapyearmtd.rb

```
def leap_year(input_year)
  ((input_year % 4 == 0)  && (input_year % 100 > 0)) ||
(input_year % 400 == 0)
end

# Get the input and determine if it is a leap year
puts "Enter the year"
STDOUT.flush
input_year = gets.chomp.to_i
if leap_year(input_year)
  puts "Year #{input_year} is a leap year and has #{366*60*24}
minutes in the year"
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

```
else
  puts "Year #{input_year} is not a leap year and has
#{365*60*24} minutes in the year"
end
```

5. Given a string **s = 'key=value'**, create two strings s1 and s2 such that s1 contains key and s2 contains value. Hint: Use some of the **String** functions. Program p021rangesex.rb

```
s = 'key=value'
i = s.index('=')
s1 = s[0...i]
puts s1
s2 = s[i+1,s.length]
puts s2
```

6. This assignment is from Chris Pine's Book.
   a. Write a Deaf Grandma program. Whatever you say to grandma (whatever you type in), she should respond with HUH?! SPEAK UP, SONNY!, unless you shout it (type in all capitals). If you shout, she can hear you (or at least she thinks so) and yells back, NO, NOT SINCE 1938! To make your program really

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

believable, have grandma shout a different year each time; maybe any year at random between 1930 and 1950. You can't stop talking to grandma until you shout BYE.

```
# p026zdeafgm1.rb
a = ( 1930...1951).to_a
puts 'Enter your response: '
STDOUT.flush
until (response = gets.chomp).eql?('BYE')
  if (response.eql?(response.upcase ))
    puts 'NO, NOT SINCE   ' + a[rand(a.size)].to_s  + '
!'
  else
    puts 'HUH?!   SPEAK UP, SONNY!'
  end
  puts 'Enter your response: '
  STDOUT.flush
end
```

b. Extend your Deaf Grandma program: What if grandma doesn't want you to leave? When you shout BYE, she could pretend not to hear you. Change your previous program so that you have to shout BYE three times in a row. Make sure to test your

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

program: if you shout BYE three times, but not in a row, you should still be talking to grandma.

```
# p026zdeafgm2.rb
a = ( 1930...1951).to_a
puts 'Enter your response: '
STDOUT.flush
until (response = gets.chomp).eql?('BYE BYE BYE')
  if response.eql?('BYE')
        # do nothing
    elsif response.eql?(response.upcase)
    puts 'NO, NOT SINCE   ' + a[rand(a.size)].to_s  + '
!'
    else
    puts 'HUH?!   SPEAK UP, SONNY!'
    end
    puts 'Enter your response: '
  STDOUT.flush
end
```

7. Write a Ruby program (call it p028swapcontents.rb) to do the following. Take two text files say A and B. The program should swap

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

the contents of A and B ie. after the program is executed, A should contain B's contents and B should contains A's.

```
# p028swapcontents.rb - Program to swap the contents of 2 text
files
# Asuumptions: The two files exist in the same folder as the
program

# Function to read contents of one file and write them to
another file
# Accepts 2 file names - file1 and file2
# Reads from file1 and writes to file2
def filereadwrite(file1, file2)
  f2 = File.open(file2, "w")
  f1 = File.open(file1, "r")
  while line = f1.gets
    f2.puts line
   end
  f1.close
  f2.close
end

filereadwrite("file1", "file1.tmp")
filereadwrite("file2", "file1")
filereadwrite("file1.tmp", "file2")

File.delete('file1.tmp')
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

8. Write a Ruby program that, when given an array as collection = [1, 2, 3, 4, 5] it calculates the sum of its elements. Program p020arraysum.rb

```
collection = [1, 2, 3, 4, 5]
sum = 0
collection.each {|i| sum += i}
puts sum
```

9. Write a Ruby program that, when given an array as collection = [12, 23, 456, 123, 4579] it displays for each number, whether it is odd or even. Program p021oddeven.rb

```
collection = [12, 23, 456, 123, 4579]
collection.each do |i|
  if i % 2 == 0
    puts "#{i} is even"
  else
    puts "#{i} is odd"
  end
end
```

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

10. Previously you had written a program that swapped the contents of two text files. Modify that program to include exception handling.

11. Write a test suite for a simple class called Student. This class stores a first name, a last name, and an age: a person's full name is available as a computed value.

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Appendix C

Over a period of time, I spoke to various Ruby / Rails Gurus. Here are the links.

David Hansson
Bruce Tate
Elliot Temple
Peat Bakke
Phil Tomson
Topher Cyll
James Gray II
Chris Anderson
Peter Cooper
Pat Eyler
Jens-Christian Fischer
Shashank Date
Chang Sau Sheong
Jeremy Voorhis

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved

Charles Nutter
Rida Al Barazi
Dalibor Sramek
Duncan Beevers
Premshree S. Pillai
Yogi Kulkarni
Olle Jonsson
Roberto Nogueira
Robert Evans
Manik Juneja
Kevin Marshall
Obie Fernandez
Karmen Blake
Francisco Alves Cabrita

Copyright 2007 by http://rubystudynotes.com/ - All Rights Reserved