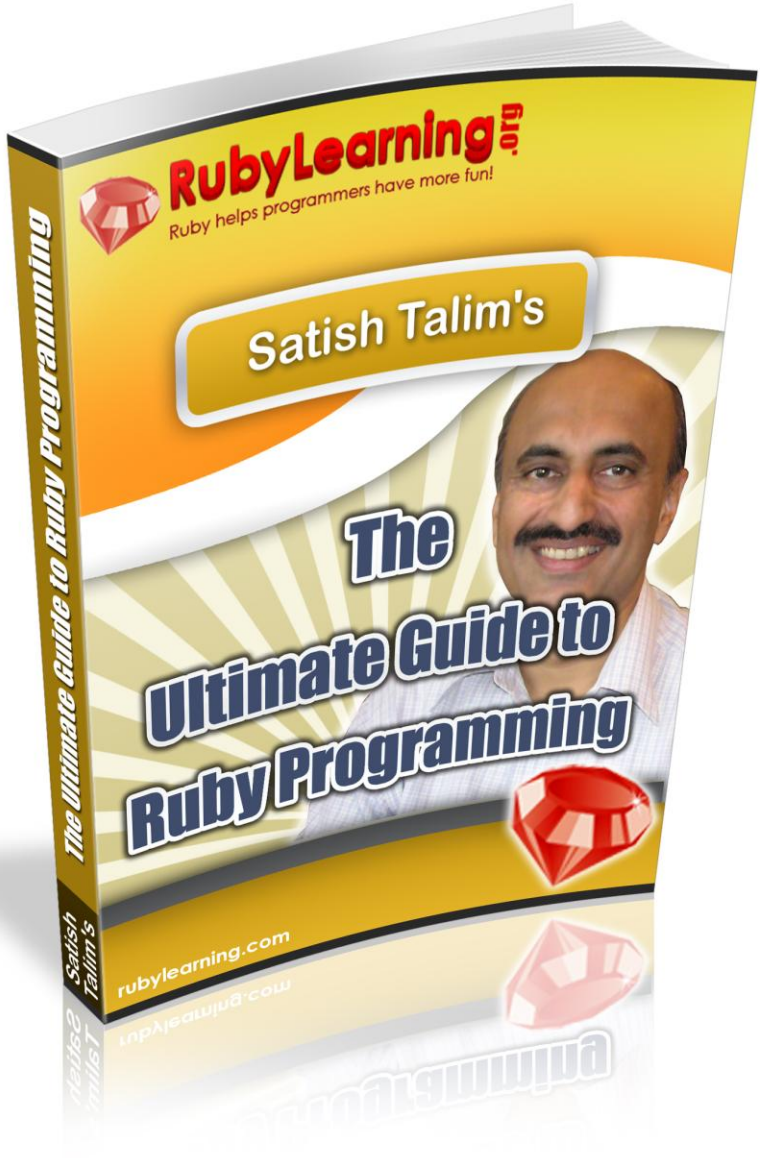


Now covers Ruby 1.9

The Ultimate Guide to Ruby Programming

Warning and Disclaimer
Every effort has been made to make this book as complete and as accurate as possible, but no warranty of fitness is implied. The information provided is on an "as is" basis. The author shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Revised Edition – 1st Aug. 2010
First Edition – June 2006

PLEASE SUPPORT RUBYLEARNING.COM

Table of Contents

Acknowledgements

My interest in Ruby was aroused after I read an article **Ruby the Rival** http://www.onjava.com/pub/a/onjava/2005/11/16/ruby-the-rival.html
in November 2005. I decided to learn Ruby myself and started making my Ruby Study Notes. What's presented here is a result of that.

There are a good number of people who deserve thanks for their help and support they provided, either while or before this eBook was written, and there are still others whose help will come after the eBook is released.

I'd like to thank everyone on the **ruby-talk** mailing list for their thoughts and encouragement; all of my wonderful **PuneRuby** http://tech.groups.yahoo.com/group/puneruby/
RUG members and my 'gang' of patrons, mentors and assistant teachers at http://rubylearning.org/ for their help in making these study notes far better than I could have done alone.

I have made extensive references to information, related to Ruby, available in the public domain (wikis and the blogs, articles of various **Ruby Gurus**).

Much of the material on **rubylearning.com** and in the course at **rubylearning.org** is drawn **primarily** from the ***Programming Ruby*** book http://pragprog.com/titles/ruby3/programming-ruby-3
available from **The Pragmatic Bookshelf**.

The following books have also been referred to:
The Ruby Programming Language
http://oreilly.com/catalog/9780596516178/,
Beginning Ruby
http://www.apress.com/book/view/9781590597668,
Learn to Program
http://www.pragprog.com/titles/fr_ltp/learn-to-program,
Ruby Cookbook

http://oreilly.com/catalog/9780596523695/
and
The Ruby Way
http://www.informit.com/store/product.aspx?isbn=0672328844;

My acknowledgment and thanks to all of them.

Introduction

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.

Concept and Approach

I have tried to organize this eBook in which each chapter builds upon the skills acquired in the previous chapter, so that you will never be called upon to do something that you have not already learned. This eBook not only teaches you how to do something, but also provides you with the chance to put those morsels of knowledge into practice with exercises. I have therefore included several exercises, or assignments, in this eBook so that you will have opportunities to apply your knowledge.

How to Use This Ruby eBook

I recommend that you go through the entire Ruby eBook chapter by chapter, reading the text, running the sample programs and doing the assignments along the way. There are no large applications in this book – just small, self-contained sample programs. This will give you a much broader understanding of how things are done (and of how you can get things done), and it will reduce the chance of anxiety, confusion, and worse yet, mistakes.

It is best to read this eBook and complete its assignments when you are relaxed and have time to spare. Nothing makes things go wrong more than working in a rush. And keep in mind that Ruby and the assignments in this eBook are fun, not just work exercises. So go ahead and enjoy it.

About the Conventions Used in This Ruby eBook

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

IN RAILS: This explains how the relevant Ruby topic is being used in Rails.

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

```
def hello
  puts "hello"
end
```

We shall be using Ruby for the language, **`ruby`** for the interpreter.

Ruby Code layout is pretty much up to you; indentation is not significant (but using two-character indentation will make you friends in the Ruby community if you plan on distributing your code).

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.

For more information on **Ruby coding style**, please refer to –

The Unofficial Ruby Usage Guide – http://www.caliban.org/ruby/rubyguide.shtml

Ruby 101: Naming Conventions – http://www.softiesonrails.com/2007/10/18/ruby-101-naming-conventions

Ruby Coding Convention –

http://ruby-programming.learnhub.com/lesson/page/5028-ruby-coding-convention

When there is a sample program to accompany the code, the program name is shown like this: **hello.rb**

Though we would be discussing Ruby 1.9.1 on the Windows platform, these notes are appropriate for Linux/Mac users as well.

You can cut-paste all of the Ruby programs shown in this Ruby eBook, from the following link:
http://rubylearning.com/satishtalim/course_ruby_programs.html

Please remember that in Ruby, there's always more than one way to solve a given problem.

If you notice any errors or typos, or have any comments or suggestions or good exercises I could include, please email me at **satish.talim@gmail.com**.

Learn Core Ruby Online

If you want to learn Core Ruby Programming (FORPC101), do join the batch at http://rubylearning.org/

Over 25000 people from across the globe (from over 140 countries) have either already learnt or are in the process of learning Core Ruby from that site.

RubyLearning is the first and only site in the world that teaches Core Ruby Programming for free. Over 30 Assistant Teachers, Mentors and Patrons help you through the learning process – 24x7.

Installation

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.

Topher Cyll (author of *Practical Ruby Projects*) says that *Ruby is a wonderful language for hacking, design, and programming, not to mention an excellent tool for scripting, text-processing, and system administration. Combined with the Ruby on Rails web development buzz, Ruby's future is promising, particularly with progress toward a faster runtime environment.*

Yukihiro Matsumoto, commonly known as 'Matz' created the Ruby language in 1993. An **interview** with him in 2001, http://www.linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html talks about the history of Ruby. Matz's "**The Philosophy of Ruby**" http://www.artima.com/intv/ruby.html is another good read.

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

Ruby Community

Again, Topher Cyll says that *Ruby's strength is also its community. Now, every language community has its own flavor and culture. Maybe it is just because it's a fresh language with the right set of features, but the programmers you meet in the Ruby track at conferences, the hackers at your local Ruby Brigade, and the guy down the hall at work sneaking Ruby into the system all seem to have something in common. They're curious, reflective, and lighthearted, but they're also highly effective programmers. And they're all working on some kind of project. It'll be born of personal interest, but odds are it will be shared—and adopted. That's just the community standard around here!*

Downloading Ruby

As an open source language, Ruby has been ported to run on many different computer platforms and architectures. This means that if you develop a Ruby program on one machine, it's likely you'll be able to run it without any changes on a different machine. You can use Ruby, in one form or another, on most operating systems and platforms.

Though I talk about Ruby on a Windows platform, **this eBook is appropriate for Linux/Mac users as well**.

The simplest way to get Ruby installed on a PC is by using the *Ruby Installer for Windows*.

http://rubyinstaller.org/

Download the latest version of Ruby 1.9. After you have 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. *Always check whether this has been set properly or not. If not, set it externally.*

You can use any editor you like, as long as it's a plain-text editor and not a word processor) and a directory (a.k.a. a folder) in which to store your Ruby program files. The folder is separate from other work areas so that you can keep track of your practice program files.

Geany
http://geany.uvena.de/
is another editor that you can explore. It is small and lightweight with an integrated development environment. It was developed to provide a small and fast IDE, which has only a few dependencies from other packages. Geany is known to run under Linux, FreeBSD, NetBSD, MacOS X, AIX v5.3, Solaris Express and Windows. More generally, it should run on every platform, which is supported by the GTK libraries.

You'll find two executables in the Ruby Windows distribution. ***ruby.exe*** is meant to be used at a command prompt (a DOS shell), just as in the Unix version. For applications that read and write to the standard input and output, this is fine. But this also means that anytime you run *ruby.exe*, you'll get a DOS shell even if you don't want one - Windows will create a new command prompt window and display it while Ruby is running. This may not be appropriate behaviour if, for example, you double-click a Ruby script that uses a graphical interface (such as Tk), or if you are running a Ruby script as a background task or from inside another program. In these cases, you'll want to use ***rubyw.exe***. It is the same as *ruby.exe* except that it does not provide standard in, standard out, or standard error and does not launch a DOS shell when run.

Do note that these instructions *assume* that you are going to use a Windows platform. For installation on other platforms, you can refer:
http://www.ruby-lang.org/en/downloads/

Ruby Programming Environment

Let's assume that you have ***Windows operating system*** and have installed Ruby in the folder **c:\ruby**, then the installation creates a number of sub-folders. Some of the folders are:
c:\ruby\bin is where the Ruby executables (including ruby and irb) have been installed.
c:\ruby\lib\ruby\1.9.1 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\1.9.1\i386-mingw32 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 find (possibly among other things) the source code for the Rails framework.

Our First Ruby program

Let's open up our plain-text editor. As far as possible, ensure that your editor's Tab is set to 2 spaces. We are now ready to write our first Ruby program.

Code layout is pretty much up to you; indentation is not significant (but using two-character indentation will make you friends in the community if you plan on distributing your code).

Create a folder 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**

By convention, Ruby source files have the .rb file extension. In Microsoft Windows, Ruby source files sometimes end with .rbw, as in myscript.rbw. 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

Type the following in your editor:

```
puts 'Hello'
p 'hello'
print 'hello'
```

and then click *File/Save As...* Give the name **p001hello.rb** and store it in your **rubyprograms** folder. To run your program, open a command window and type the following as shown in bold:

```
c:\rubyprograms> ruby p001hello.rb
Hello
"hello"
hello
c:\rubyprograms>
```

You should see the output as shown above.

Note: Ruby is a scripting language. There is no special main method in Ruby from which execution begins. The Ruby interpreter is given a script of statements to execute, and it begins executing at the first line and continues to the last line. **`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. We shall talk about **p** and **print**, later on.

a. Parentheses are usually optional with a method call. These calls are all valid:
foobar
foobar()
foobar(a, b, c)
foobar a, b, c

b. 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.

IRB – A Short Introduction

(Thanks to Victor Goff)

Using IRB with Ruby makes sense when you are trying to test ideas. To start using IRB it is pretty straight forward. You can type the command '**irb**' at the console... like this:

```
c:\> irb
>>
```

If you do not have the --simple-prompt option activated, you will see something more like this:

```
c:\> irb
irb(main):001:0>
```

We shall be using the short or '--simple-prompt' mode for this discussion.

First Exercise: Hello World!

Go ahead and type the following at your IRB prompt...

```
>> 'Hello world!'
=> "Hello world!"
```

Note that the => means what is *returned*.

If we had requested output instead of a simple return, our outcome would have been different.

```
>> puts 'Hello world!'
Hello world!
=> nil
```

So, what are we seeing here?

- Our command at the '>>' prompt. In this case the result of the method **puts** (not the return) which is a printed message to screen (**STDOUT**) of Hello World!.
- And finally we see the return of the line.

Let's get a couple other samples in here, and then it will be up to you to experiment.

```
>> 1 == 1
=> true
>> 1 == 2
=> false
>> 99 & 1
=> 1
>> 50 & 1
=> 0
```

Have fun, experiment, learn.

To exit from IRB, simply type **`exit`** at the prompt.

One last thing... if you do get stuck in here (which happens sometimes for non-matching quotes, or parenthesis, or blocks, etc....) you can try pressing CTRL-C or CTRL-D. If you are not stuck, and simply want to quit out of IRB you can enter the command '**`exit`**' without quotes.

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. 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

Question asked by you

What do you mean by "Ruby is a Dynamic programming language".
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:
http://en.wikipedia.org/wiki/Dynamic_programming_language
for more details.

Some Features of Ruby

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

1. **Free format** - You can start writing your program from any line and column.
2. **Case sensitive -** Lowercase letters and uppercase letters are distinct. The keyword **`end`**, for example, is completely different from the keyword **`END`**.
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`** This only works if the = signs are the first characters of each line.
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
5. **Keywords** - Also known as reserved words (around 41 of them) in Ruby typically cannot be used for other purposes. In addition to these keywords, there are three keyword-like tokens that are treated specially by the Ruby parser when they appear at the beginning of a line: **`=begin`**, **`=end`**, **`__END__`** You may be used to thinking that a false value may be represented as a zero, a null string, a null character, or various other things. But in Ruby, all of these *values* are true; in fact, *everything is true* except the reserved words **`false`** and **`nil`**. Keywords would be called "reserved words" in most languages and they would never be allowed as identifiers. The Ruby parser is flexible and does not complain if you prefix these keywords with @, @@ or $ prefixes and use them as instance, class or global variable names. The best practice is to treat these keywords as reserved.
6. **Program encoding** - At the lowest level, a Ruby program is simply a sequence of characters. Ruby's lexical rules http://en.wikipedia.org/wiki/Lexical_analysis

are defined using characters of the ASCII character set. All Ruby keywords are written using ASCII characters, and all operators and other punctuation are drawn from the ASCII character set.

Documentation - The complete documentation for Ruby is available online here:
http://www.ruby-doc.org/

The Ruby Cheat Sheet is here:
http://cheat.errtheblog.com/

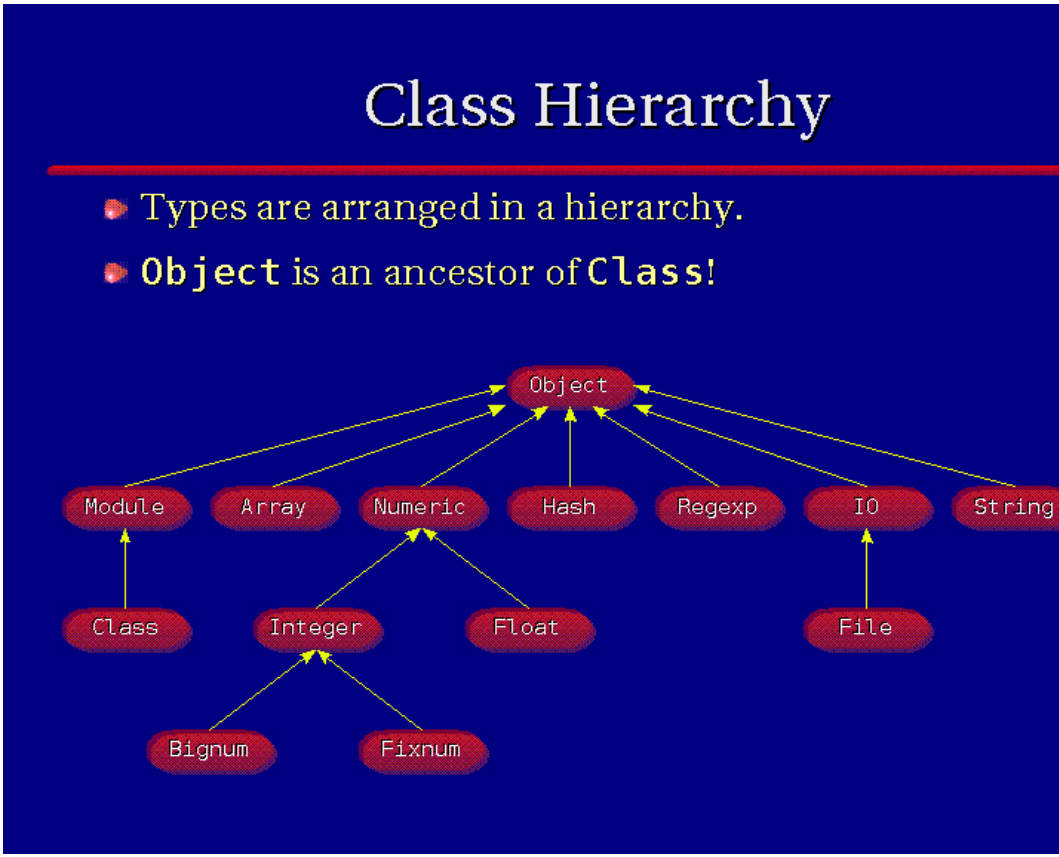
Numbers in Ruby

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). An integer literal is simply a sequence of digits eg. 0, 123, 123456789. Underscores may be inserted into integer literals (though not at the beginning or end), and this feature is sometimes used as a thousands separator eg. 1_000_000_000. Underscore characters are ignored in the digit string. Here's 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
puts 1.5 / 2.6
```

Ruby integers are objects of class **Fixnum** or **Bignum**. The **Fixnum** and **Bignum** classes represent integers of differing sizes. Both classes descend from **Integer** (and therefore **Numeric**). The floating-point numbers are objects of class **Float**, corresponding to the native architecture's double data type. The **Complex**, **BigDecimal**, and **Rational** classes are not built-in to Ruby but are distributed with Ruby as part of the standard library. We shall be talking about classes in detail later.

The class hierarchy is as shown in this figure:



The above figure courtesy Donald Craig
http://www.cs.mun.ca/~donald/

Operators and Precedence

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

Table 21.4. Ruby operators (high to low precedence)

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

a. The increment and decrement operators (++ and --) are not available in Ruby, neither in "pre" nor "post" forms. However, do note that the += and -= are available.
b. Brackets (parentheses) work the same way as with regular arithmetic. Anything inside brackets is calculated first (or, more technically, given higher precedence).
c. The check marked operators is a kind of syntactic sugar (more on this later) - where something looks like an operator but is a method call.

The Ruby modulus operator's (%) behavior is as follows:

```
puts (5 % 3)      # prints  2
puts (-5 % 3)     # prints  1
puts (5 % -3)     # prints -1
puts (-5 % -3)    # prints -2
```

Ruby's definition of the modulo (%) operator
http://en.wikipedia.org/wiki/Modulo_operation
differs from that of C and Java. In Ruby, -7%3 is 2. In C and Java, the result is -1 instead. In Ruby, **the sign of the result (for % operator) is always the same as the sign of the second operand.**

Some Technical Explanation for % operator

Given (-n % x) or (n % -x), the RHS (right hand side) of the modulus operation must be multiplied by the *negative* integer closest to 0 that will get it *past* the number on the LHS (left hand side). Then, the result is the difference between this number and the LHS. e.g.:

```
-7 % 3   # result is 2
# climbing (positive) from -9 (3 * -3) to -7

7 % -3   # result is -2
# dropping (negative) from 9 (-3 * -3) to 7
```

Of course, when the polarity is the same on both sides, such as (n % x) or (-n % -x), you just multiply the RHS times the *positive* integer that gets you the closest to the RHS *without* going past it then the result is the difference between the remainder and the LHS. e.g.:

```
7 % 3    # result is 1
# climbing (positive) from 6 (3 * 2) to 7

-7 % -3  # result is -1
# dropping (negative) from -6 (-3 * 2) to -7
```

It might help to note that in Ruby, the RHS always dictates the polarity of the result.

Difference between or and || operator.

Both **or** and **||** return their first argument unless it is **false**, in which case they evaluate and return their second argument. This is shown in the following example:

```
puts nil || 2008
puts false || 2008
puts "ruby" || 2008
```

The output is:

```
>ruby test.rb
2008
2008
ruby
>Exit code: 0
```

The only difference between **or** and **||** is their precedence. **||** has a higher precedence than **or**.

A common idiom is to use **||** to assign a value to a variable only if that variable isn't already set. This can be written as:

```
@variable = @variable || "default value"
```

or, more idiomatically, as:

```
@variable ||= "default value"
```

One reason for these alternate versions of the Boolean operators is the fact that they have lower precedence than the assignment operator. This means that you can write a Boolean expression such as the following that assigns values to variables until it encounters a false value:

```
if a = f(x) and b = f(y) and c = f(z) then d =
g(a,b,c) end
```

This expression simply would not work if written with **&&** instead of **and**.

Suggestions:

Hear Geoff Grosenbach's Ruby Basics
http://media.libsyn.com/media/carsonsystems/Vitamin_Training_-_Geoff_Grosenbach_01.mp3

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.

```
# Ruby Strings
=begin
  Ruby Strings
  In Ruby, strings are mutable
=end

puts "Hello World"
# Can use " or ' for Strings, but ' is more efficient
puts 'Hello World'
# String concatenation
puts 'I like' + ' Ruby'
# Escape sequence
puts 'It\'s my Ruby'
# New here, displays the string three times
puts 'Hello' * 3
# Defining a constant
# More on Constants later, here
# http://rubylearning.com/satishtalim/ruby_names.html
PI = 3.1416
puts PI
```

a. If **puts** is passed an object that is not a string, **puts** calls the **to_s** method of that object and prints the string returned by that method.
b. In Ruby, strings are **mutable**. They can expand as needed, without using much time and memory. Ruby stores a string as a sequence of characters.

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

```
puts `dir`
```

The command output string is sent to the operating system as a command to be executed (under windows operating system, we have sent the dir command), whereupon the output of the command (dir on a command window would display all the files and sub directories in your folder) is displayed by **puts**.

On Linux and Mac, you can instead do:

```
puts `ls`
```

Another way to spawn a separate process is to use the **Kernel** method **system**. The method executes the given command in a sub-process; it returns **true** if the command was found and executed properly. It returns **false** if command exited with a nonzero exit status, and **nil** if the command failed to execute. Remember, the command's output will simply go to the same destination as your program's output.

```
system("tar xzf test.tgz") # => true
```

Variables and Assignment

A *bareword* is any combination of letters, numbers, and underscores, and is not qualified by any symbols (**Reference**: http://alumnus.caltech.edu/~svhwan/prodScript/avoidBarewords.html) . Local variables and *barewords* look similar; they must start with either the underscore character (_) or a lowercase letter, and they must consist entirely of letters, numbers, and underscores. Remember, local variable references look just like method invocation expressions and *Keywords* can't be used as variable names.

Method calls can also be barewords, such as my_method. **gets** is a method call; so is **system**. Whenever Ruby sees a bareword, it interprets it as one of three things: (a) If there's an equal sign (=) to the right of the bareword, it's a *local variable* undergoing an assignment. (b) Ruby has an internal list of keywords and a bareword could be a *keyword*. (c) If the bareword is not (a) or (b), the bareword is assumed to be a *method call*. If no method by that name exists, Ruby raises a **NameError**.

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
```

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

```
# p004stringusage.rb
# Defining a constant
PI = 3.1416
puts PI
# Defining a local variable
my_string = 'I love my city, Pune'
puts my_string
=begin
  Conversions
  .to_i, .to_f, .to_s
=end
var1 = 5;
var2 = '2'
puts var1 + var2.to_i
# << appending to a string
a = 'hello '
a<<'world.
I love this world...'
puts a

=begin
  << marks the start of the string literal and
  is followed by a delimiter of your choice.
  The string literal then starts from the next
  new line and finishes when the delimiter is
  repeated again on a line on its own. This is
  known as Here document syntax.
=end
a = <<END_STR
This is the string
And a second line
END_STR
puts a
```

In the example:

```
x = "200.0".to_f
```

the dot means that the *message* "to_f" is being *sent* to the string "200.0", or that the method to_f is being called on the string "200.0". The string "200.0" is called the *receiver* of the message. Thus, when you see a dot in this context, you should interpret it as a message (on the right) is being sent to an object (on the left).

Some useful Ruby methods – 1

printf

Another output method we use a lot is **printf**, which prints its arguments under the control of a format string (just like printf in C or Perl).

```
printf("Number: %5.2f,\nString: %s\n", 7.18,
"FORPC101")
```

The output is:

```
>ruby tmp2.rb
Number: 7.18,
String: FORPC101
>Exit code: 0
```

In this example, the format string "Number: %5.2f,\nString: %s\n" tells **printf** to substitute in a floating-point number (allowing five characters in total, with two after the decimal point) and a string. Notice the newlines (\n) embedded in the string; each moves the output onto the next line.

step

Class **Numeric** also provides the more general method **step**, which is more like a traditional for loop.

```
8.step(40, 5) {|i| print i, " " }
```

This produces an output as:

```
>ruby tmp2.rb
8 13 18 23 28 33 38
>Exit code: 0
```

```
exit( true | false | status=1 )
```

The method **exit** terminates your program, returning a status value to

the operating system. The optional parameter is used to return a status code to the invoking environment. With an argument of **true**, exits with a status of zero. With an argument that is **false** (or no argument), exits with a status of 1, otherwise exits with the given status.

For example:

```
exit 99
```

will exit with a status of 99.

local_variables

This returns the names of the current local variables in a particular context. It is a reflective method and can be placed anywhere in the program ie. even on line 1. In the example:

```
a = 10
def mtd
 b = 20
end
puts local_variables
```

The output is:

```
>ruby tmp2.rb
a
>Exit code: 0
```

Summary

The summary of what we have read so far:

1. We are discussing Ruby 1.9 on the Windows platform. This course is appropriate for Linux/Mac users as well.
2. Ruby is an interpreted language
3. In Ruby, there's always more than one way to solve a given problem.
4. The code examples would be run in the SciTE editor and ensure that you have done the relevant settings as mentioned on the page "First Ruby program".
5. Code layout is pretty much up to you; indentation is not significant (but using two-character indentation will make you friends in the community if you plan on distributing your code).
6. By convention, Ruby source files have the **.rb** file extension. In Microsoft Windows, Ruby source files sometimes end with .rbw, as in myscript.rbw.
7. In Ruby, program execution proceeds in general from top to bottom.
8. Features: Free format, Case sensitive, Two type of comments, Statement delimiters are not required, Around 41 Keywords, and all Ruby keywords are written using ASCII characters, and all operators and other punctuation are drawn from the ASCII character set.
9. You may be used to thinking that a false value may be represented as a zero, a null string, a null character, or various other things. But in Ruby, all of these *values* are true; in fact, everything is true except the reserved words false and nil.
10. We shall be referring to the documentation here.

11. **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.

12. Parentheses are usually optional with a method call. These calls are all valid: foobar foobar() foobar(a, b, c) foobar a, b, c

13. 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).

14. Some very common Ruby operators:+ addition; - subtraction; * multiplication; / division

15. The increment and decrement operators (++ and --) are not available in Ruby, neither in "pre" nor "post" forms.

16. Anything inside brackets is calculated first (or, more technically, given higher precedence).

17. Observe how the modulus operator (%) works in Ruby.

18. When you do arithmetic with integers, you'll get integer answers.

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

20. In Ruby, strings are mutable. They can expand as needed, without using much time and memory.

21. String concatenation is joining of two strings, using the + operator.

22. Escape sequence is the \ character. Examples: \", \\, \n

23. '' is an empty string.

24. If you get a compilation error like - #<TypeError: cannot convert Fixnum into String> it means that you cannot really add a number to a string, or multiply a string by another string.

25. Constants begin with capital letters. Example PI, Length

26. A variable springs into existence as soon as the interpreter sees an assignment to that variable. It is a good practice to assign `nil` to a local variable initially, otherwise a runtime error will be generated if the local variable is used without being assigned a value.

27. x, y = y, x will interchange the values of x and y. Parallel assignment is any assignment expression that has more than one lvalue, more than one rvalue, or both. Multiple lvalues and multiple rvalues are separated from each other with commas.

28. Local variables must start with either a lowercase letter or the underscore character (_), and they must consist entirely of letters, numbers, and underscores. Examples: india, _usa, some_var

29. .to_i, .to_f, .to_s are used to convert to an integer, float, string respectively.

30. The operator << is used to append to a string

Exercise Set 1

1. This is a sample question from the "Ruby Association Certified Ruby Programmer" examination. Thanks to Satoshi Asakawa for the Japanese to English translation.

Select all correct answers which outputs Error message.
Answers:

```
2 + 8
3 * 10
4 ** 10
"abcde" * 3
"abcde" + 1
"abcde" + "fghij"
```

2. Before executing the code given below, guess the results. Next, execute the code. Did you get it right? If you did not get it right, can you think of why?
Goal: Understanding operator precedence and association.

```
y = false
z = true
x = y or z
puts x
(x = y) or z
puts x
x = (y or z)
puts x
```

3. Read up the **`sprintf`** documentation
http://ruby-doc.org/core/classes/Kernel.html#M005983
and the **%** documentation in the **String** class and figure out the output being printed by this Ruby code. Explain why you get this output.
Goal: To realize that the Ruby documentation is at times incomplete or not clear.

```
puts "%05d" % 123
```

4. Write a Ruby program that displays how old I am, if I am 979000000 seconds old. Display the result as a floating point (decimal) number to two decimal places (for example, 17.23).
Note: 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

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

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 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

Note: Do not worry if you do not understand this, right now. 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.

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 (on the right side of SciTE)** 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 (note that this is Ruby internal buffering only; the OS may buffer the data as well). The usage is not mandatory but recommended.

gets accepts a single line of data from the standard input - the keyboard in this case - and assigns the string typed by the user to city. The standard input is a default stream supplied by many operating systems that relates to the standard way to accept input from the user. In our case, the standard input is the keyboard.

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'.

In Rails: 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.

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" through "z". Uppercase letter means "A" through "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 (declared within an object) name consists of a lowercase letter (or an underscore) followed by name characters (sunil, _z, hit_and_run).

2.2. An **instance** variable (declared within an object and always "belongs to" whatever object **self** refers to) name starts with an "at" sign ("@") followed by a *name* (@sign, @_, @Counter).

2.3. A **class** variable (declared within a class) name starts with two "at" signs ("@@") followed by a *name* (@@sign, @@_, @@Counter). These are rarely used in Ruby programs. A class variable is shared among all objects of a class. Only one copy of a particular class variable exists for a given class. Class variables used at the top level are defined in **Object** and behave like global variables

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. Examples: 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). Ruby defines a number of global variables that include other punctuation characters, such as $_ and $-K.

3. **Method** names should begin with a lowercase letter (or an underscore). “?”, “!” 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. More on Bang methods later - http://rubylearning.com/satishtalim/writing_own_ruby_methods.html)

The *Ruby convention* is to use underscores to separate words in a multiword method or variable name. By convention, most constants are written in all uppercase with underscores to separate words, LIKE_THIS. Ruby class and module names are also constants, but they are conventionally written using initial capital letters and camel case, LikeThis. 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 on constants later - http://rubylearning.com/satishtalim/ruby_constants.html). 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**

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

# In Ruby, everything you manipulate is an object
'I love Ruby'.length
```

The basic types in Ruby are **Numeric** (subtypes include **Fixnum, Integer**, and **Float**), **String, Array, Hash, Object, Symbol, Range**, and **Regexp**.

Though we have not learned classes yet, nevertheless here are some more details about a class called **Float**.

Float is a sub class of **Numeric**.

Float objects represent real numbers using the native architecture's double-precision floating point representation.

DIG is a class constant that gives precision of **Float** in decimal digits. On my PC, the code:

```
puts Float::DIG
```
outputs 15.

MAX is another class constant that gives the largest **Float.** For example:

```
puts Float::MAX
```
outputs 1.79769313486232e+308

Let us look at Peter Cooper's example in his *Beginning Ruby* book (never mind if you do not follow the code yet):

```
rice_on_square = 1
64.times do |square|
  puts "On square #{square + 1} are #{rice_on_square}
grain(s)"
  rice_on_square *= 2
end
```

By square 64, you're up to placing many trillions of grains of rice on each square!

It proves that Ruby is able to deal with extremely large numbers, and unlike many other programming languages, there are no inconvenient limits. Ruby does this with different classes, one called **Fixnum** (default) that represents easily managed smaller numbers, and another, aptly called **Bignum**, that represents "big" numbers Ruby needs to manage internally. Ruby will handle Bignums and Fixnums for you, and you can perform arithmetic and other operations without any problems. Results might vary depending on your system's architecture, but as these changes are handled entirely by Ruby, there's no need to worry.

Ruby doesn't require you to use primitives (data types)
http://en.wikipedia.org/wiki/Primitive_type

http://en.wikipedia.org/wiki/Data_type
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. The class **Object** has a method called **class** that returns the class of an object, for example:

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

Another example (do not worry, if you don't understand the code now):

```
puts 'I am in class = ' + self.class.to_s
puts 'I am an object = ' + self.to_s
print 'The object methods are = '
puts self.private_methods.sort
```

We shall talk about **self** later on.
http://rubylearning.com/satishtalim/ruby_self.html
private_methods is a method of the **Object** class and **sort** is a method of the **Array** class.

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
```

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
```

We shall be talking about **self** later on, here:
http://rubylearning.com/satishtalim/ruby_self.html

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 statement executed in the method. 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
  'Hello'
end
#use the method
Puts hello

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

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

The output is:

```
>ruby p008mymethods.rb
Hello
Hello satish
Hello talim
>Exit code: 0
```

Please note that from 1.9, you can have unparenthesized arguments to methods.

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
```

Please note that as of now, there is no way, to specify a value for the second parameter and use the default value of the first parameter.

In the above program the interpolation operator #{...} gets calculated separately 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.

Note: Interpolation refers to the process of inserting the result of an expression into a string literal. The way to interpolate within a string is to place the expression within #{ and } symbols. An example demonstrates this:

```
puts "100 * 5 = #{100 * 5}"
```

This displays:

```
100 * 5 = 500
```

The #{100 * 5} section interpolates the result of 100 * 5 (500) into the string at that position, resulting in the output shown.

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
```

The output is:

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

alias creates a new name that refers to an existing method, operator, global variable, or regular expression back-reference ($&, $`, $', and $+). Local variables, instance variables, class variables, and constants may not be aliased. The parameters to alias may be names or symbols.

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|
    words
  end
end
puts foo('hello','world')
puts 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 which we talk in length here.
http://rubylearning.com/satishtalim/ruby_blocks.html
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.

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?
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?
Observe the following example:

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

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

Gary Wright in the Ruby forum posted in reply to some posts: "*It is confusing to me to even think about methods returning objects unless you are using that as a very specific shorthand for saying that methods return *references* to objects. That is the unifying idea that helped me understand how Ruby manipulates data -- it is all references and not*

the objects themselves. The objects themselves are almost completely hidden from the programmer (excluding C extensions) in Ruby. Everything is a reference to an object."

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 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.

In Ruby you can define a method name that ends with an exclamation point or bang. The bang methods are called and executed just like any other method. However, by convention, a method with an exclamation point or bang is considered dangerous.

Normally for the built-in classes, dangerous usually (although not always) means this method, unlike its non-bang equivalent, permanently modifies its receiver.

You'll find a number of methods, one with the bang and one without. Those without the bang perform an action and return a new object. 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.

A few non-bang methods perform changes on the original string. The names of these methods make it clear that this is happening (such as **replace**), even though there's no ! on the name.

Method names ending with ?

The question mark has no special meaning to the Ruby interpreter. However, by convention, any method whose name ends with ? returns a value that answers the question posed by the method invocation. The **empty?** method of an array, for example, returns **true** if the array has no elements. Mostly such methods return one of the Boolean values **true** or **false**, but this is not required, as any value other than **false** or **nil** works like **true** when a Boolean value is required. The **Numeric** method **nonzero?**, for example, returns **nil** if the number it is invoked on is zero, and just returns the number otherwise.

Summary

1. Avoid using Global scope and Global Variables. Global scope means scope that covers the entire program. Global variables are distinguished by starting with a dollar-sign ($) character. The Ruby interpreter starts up with a fairly large number of global variables already initialized.

2. **gets** (get a string) and **chomp** (a string method) are used to accept input from a user.

3. **gets** returns a string and a '\n' character, while **chomp** removes this '\n'.

4. **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.

5. To format the output to say 2 decimal places, we can use the **Kernel**'s **format** method.

6. 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.

7. 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".

8. 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.

9. You can use variables in your Ruby programs without any declarations. Variable name itself denotes its scope (local, global, instance, etc.).

10. **REMEMBER** the way local, instance, class and global variables, constants and method names are declared.

11. "?", "!" and “=” are the only weird characters allowed as method name suffixes.

12. The Ruby convention is to use underscores to separate words in a multiword method or variable name. By convention, most constants are written in all uppercase with underscores to separate words, LIKE_THIS. Ruby class and module names are also constants, but they are conventionally written using initial capital letters and camel case, LikeThis. More examples: my_variable, MyModule, MyClass, My_Constant.

13. Any given variable can at different times hold references to objects of many different types.

14. Variables in Ruby act as "references" to objects, which undergo automatic garbage collection.

15. For the time being, remember that Ruby is dynamically typed and that in Ruby, everything you manipulate is an object and the results of those manipulations are themselves objects.

16. The basic types in Ruby are **Numeric** (subtypes include **Fixnum**, **Integer**, and **Float**), **String**, **Array**, **Hash**, **Object**, **Symbol**, **Range**, and **Regexp**.

17. For the time being, remember that you can always see what object you are in (current object) by using the special variable **self**.

18. We use **def** and **end** to declare a method. Parameters are simply a list of local variable names in parentheses.

19. We do not declare the return type; a method returns the value of the last statement.

20. It is recommended that you leave a single blank line between each method definition.

21. Ruby allows parentheses to be omitted from most method declarations and / or invocations. In simple cases, this results in

clean-looking code. In complex cases, however, it causes syntactic ambiguities and confusion.

22. Methods that act as queries are often named with a trailing ?

23. Methods that are "dangerous," or modify the receiver, might be named with a trailing ! (Bang methods)

24. 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.

25. For now remember that there is an interpolation operator #{...}

26. **alias** 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.

27. In Ruby, we can write methods that can accept variable number of parameters.

28. There's no limit to the number of parameters one can pass to a method.

29. The sequence in which the parameters are put on to the stack are left to right.

30. Whether Ruby passes parameters by value or reference is very debatable - it does not matter.

Exercise Set 2

1. The following program prints the value of the variable. Why?

```
my_string = 'Hello Ruby World'
def my_string
 'Hello World'
end
puts my_string
```

2. Write a method called **convert** that takes one argument which is a temperature in degrees Fahrenheit. This method should return the temperature in degrees Celsius. The program should display the temperature in degrees Celsius to 2 decimal places. 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 the string 45.57. Another way is to use the **round** function as follows: puts (x*100).round/100.0

ri and RDoc

If you have a good internet connection, then you would probably refer to the Ruby documentation online.
http://www.ruby-doc.org/ruby-1.9/index.html

However, for those with a slower connection or not having an internet access, the Ruby **ri** and **RDoc** tools are very useful.

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.

The Ruby **ri** tool is used to view the Ruby documentation off-line. Open a command window and invoke **ri** followed by the name of a Ruby class, module or method. **ri** will display documentation for you. You may specify a method name without a qualifying class or module name, but this will just show you a list of all methods by that name (unless the method is unique). Normally, you can separate a class or module name from a method name with a period. If a class defines a class method and an instance method by the same name, you must instead use :: to refer to a class method or # to refer to the instance method. Here are some example invocations of ri:

```
ri Array
ri Array.sort
ri Hash#each
ri Math::sqrt
```

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:

http://www.caliban.org/ruby/rubyguide.shtml#ri
http://en.wikipedia.org/wiki/RDoc
http://www.ruby-doc.org/core/classes/RDoc.html

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.

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. The backslash works to escape another backslash, so that the second backslash is not itself interpreted as an escape character. In single-quoted strings, a backslash is not special if the character that follows it is anything other than a quote or a backslash. For example 'a\b' and 'a\\b' are equal. 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.

```
# p013expint.rb
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
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,

http://www.rubyist.net/~slagell/ruby/strings.html

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
```

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'
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
```

Using %w

Sometimes creating arrays of words can be a pain, what with all the quotes and commas. Fortunately, Ruby has a shortcut: %w does just what we want.

```
names1 = [ 'ann', 'richard', 'william', 'susan', 'pat' ]
puts names1[0] # ann
puts names1[3] # susan
# this is the same:
names2 = %w{  ann richard william susan pat }
puts names2[0] # ann
puts names2[3] # susan
```

See the Complete String documentation:
http://www.ruby-doc.org/ruby-1.9/classes/String.html

Simple Constructs in Ruby

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
```

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!!'
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: ==, != >=, <=, >, <

unless, as a statement or a modifier, is the opposite of **if**: it executes code only if an associated expression evaluates to **false** or **nil**. Ruby's **unless** construct begins with **unless** and ends with **end**. The body is the text between the two.

```
unless ARGV.length == 2
  puts "Usage: program.rb 23 45"
  exit
end
```

In the above program, the body is executed unless the number of elements in the array is equal to 2 (meaning that both arguments were given). The method **Kernel.exit** terminates your program, returning a status value to the operating system.

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
```

Conditional ?:

As a concise alternative to simple if/else statements we can use the conditional or ternary **?:** operator. It is the only ternary operator (three operands) in Ruby. It has the following basic structure:

```
(condition) ? (result if condition is true) : (result if condition
is false)
```

The first operand appears before the question mark. The second operand appears between the question mark and the colon. And the third operand appears after the colon. The question mark must appear on the same line as the first argument and the colon must appear on the same line as the second argument. The **?:** operator always evaluates its first operand. If the first operand is anything other than **false** or **nil**, the value of the expression is the value of the second operand. Otherwise, if the first operand is **false** or **nil**, then the value of the expression is the value of the third operand. Let's look at an example:

```
age = 15
# We talk about the Range class later on
# will output teenager
puts (14...20).include?(age) ? "teenager" : "not a teenager"
```

The ternary operator also can be useful for conditional assignments:

```
age = 23
person = (14...20).include?(age) ? "teenager" : "not a teenager"
puts person # => "not a teenager"
```

Statement modifiers

Ruby statement modifiers are a useful shortcut if the body of an **if** or **while** statement is just a single expression. Simply write the expression, followed by **if** or **while** and the condition. For example, here's a simple **if** statement.

```
puts "Enrollments will now Stop" if participants > 250
```

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

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.

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

Some Additional Information

Though we still have to talk about classes, nevertheless here is some additional information for you.

Q. What's the difference between **FALSE** and **NIL**
A. **nil** and **false** are not the same things. Both have a false value and also remember that everything in Ruby is an object. See the following program:

```
# We can determine our object's class
# and its unique object ID
# NIL is synonym for nil
puts NIL.class # NilClass
puts nil.class # NilClass
puts nil.object_id # 4

# FALSE is synonym for false
puts FALSE.class # FalseClass
puts false.class # FalseClass
puts false.object_id # 0
```

Ruby Blocks

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. Braces have a high precedence; **do** has a low precedence. If the method invocation has parameters that are not enclosed in parentheses, the brace form of a block will bind to the last parameter, not to the overall invocation. The **do** form will bind to the invocation.

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.

Also, as you will soon learn, blocks can have their own arguments. There are many methods in Ruby that iterate over a range of values. Most of these iterators are written in such a way as to be able to take a code block as part of their calling syntax. The method can then *yield* control to the code block (i.e. execute the block) during execution as many times as is necessary for the iteration to complete (e.g. if we are iterating over array values, we can execute the block as many times as there are array values etc.).

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 a 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. Thus, any method that wants to take a block as a parameter can use the **yield** keyword to execute the block at any time.

Program **p022codeblock.rb** illustrates what we have just discussed.

```
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.** If no code block is passed, Ruby raises an exception:

```
no block given (LocalJumpError)
```

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.

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.

block_given? returns true if **yield** would execute a block in the current context. Refer to the following example:

```
def try
  if block_given?
    yield
  else
    puts "no block"
  end
end
try # => "no block"
try { puts "hello" } # => "hello"
try do puts "hello" end # => "hello"
```

Block Variables

Let us see what happens in the following example when a variable outside a block is **x** and a block parameter is also named **x**.

```
x = 10
5.times do |x|
  puts "x inside the block: #{x}"
end

puts "x outside the block: #{x}"
```

The output is:

```
x inside the block: 0
x inside the block: 1
x inside the block: 2
x inside the block: 3
x inside the block: 4
x outside the block: 10
```

You will observe that after the block has executed, **x** outside the block is the original **x**. Hence the block parameter **x** was local to the block.

Next observe what happens to **x** in the following example:

```
x = 10
5.times do |y|
  x = y
  puts "x inside the block: #{x}"
end

puts "x outside the block: #{x}"
```

The output is:

```
x inside the block: 0
x inside the block: 1
x inside the block: 2
x inside the block: 3
x inside the block: 4
x outside the block: 4
```

Since **x** is *not a block parameter* here, the variable **x** is the same inside and outside the block.

In Ruby 1.9.1, blocks introduce their own scope for the *block parameters only*. This is illustrated by the following example:

```
x = 1200
puts "Before the loop, x = #{x}"

3.times do|y;x|
  puts "Looping #{y}"
  x = y
end

puts "After the loop, x = #{x}"
```

The output is:

```
Before the loop, x = 1200
Looping 0
Looping 1
Looping 2
After the loop, x = 1200
```

In the above block, a new feature is being used: *block local variable*. In short, block local variables shield a block from manipulating variables outside of its scope. This prevents a block from unintentionally clobbering any variables outside its scope. If you don't want to clobber variables, use block local variables for the variables your block creates.

The syntax for a block local variable is simple. Put a semicolon after the normal block parameter list, then list the variable you want as block local variables. For example, if the block takes two variables a and b, and uses to local variables x and y, the parameter list would look like this: |a,b; x,y|.

Some useful Ruby methods - 2

`Method *`

`str * int --> string`

The `*` instance method of class `String`, returns a new String containing int copies of the receiver. Here's an example:

```
"Ruby! " * 3 # => "Ruby! Ruby! Ruby! "
```

`slice`

The **`String slice`** method, if passed a single **`Fixnum`**, returns the code of the character at that position (whereas in version 1.9, it returns the character at the position). If passed two **`Fixnum`** objects, returns a substring starting at the offset given by the first, and a length given by the second.

See the following examples:

```
a = "hello there"
a.slice(1) # => 101
a.slice(1,3) # => "ell"
a.slice(-3,2) # => "er"
a.slice(0) # => 104
```

`strip`

`str.strip -> string`

Returns a copy of *str* with leading and trailing whitespace removed.

```
 " hello ".strip #=> "hello"
 "\tgoodbye\r\n".strip #=> "goodbye"
```

`count`

The `String count` method counts the number of occurrences of any of a set of specified characters.

`str.count( (string)+ ) -> int`

Where (string)+ denotes comma separated string parameters.

Each string parameter defines a *set* of characters to count. The *intersection of these sets* defines the characters to count in str.

```
a = "hello world"
a.count "lo" # => 5
```

```
a.count "lo", "o" # => 2
```

length

You (or your users) frequently misremember the name of a method. To reduce the confusion, you want to make the same method accessible under multiple names. Alternatively, you're about to redefine a method and you'd like to keep the old version available.

It's difficult to pick the perfect name for a method: you must find the word or short phrase that best conveys an operation on a data structure, possibly an abstract operation that has different "meanings" depending on context. Sometimes there will be no good name for a method and you'll just have to pick one; sometimes there will be too many good names for a method and you'll just have to pick one. In either case, your users may have difficulty remembering the "right" name of the method. You can help them out by creating aliases.

Some languages use length or len to find the length of a string, and some use size.

Do you remember how you can achieve this? Of course with alias. Ruby itself uses aliases in its standard library.

```
str.length -> int
```

Returns the length of str, where str is a string.

```
str.size -> int
```

Synonym for `String#length`.

Summary

1. Refer to the **String** documentation to use the various methods available.

2. For double-quoted string literals, Ruby looks for substitutions - sequences that start with a backslash character - and replaces them with some binary value or does expression interpolation ie. within the string, the sequence #{expression} is replaced by the value of the expression.

3. 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.

4. Observe how one can list all the methods of a class or object.

5. Comparing two strings for equality can be done by == or **.eql?** (for identical content) and **.equal?** (for identical objects).

6. **%w** is a common usage in strings.

7. Observe the usage of constructs: **if else end**, **while**, **if elsif end**

8. Ruby also has a negated form of the **if** statement, the **unless end**.

9. 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. **case** returns the value of the last expression executed. Usage: **case when else end**

10. **IMPORTANT**: Ruby Code blocks are chunks of code between braces or between do- end that you can associate with method invocations.

11. Code blocks 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.

12. 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.

13. Inside a method, you can call a Ruby block using the **yield** keyword with a value.

14. 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 (|).

Exercise Set 3

1. Write a program that processes the string s = "Welcome to the forum.\nHere you can learn Ruby.\nAlong with other members.\n" a line at a time, using all that we have learned so far. The expected output is:

```
>ruby tmp.rb
Line 1: Welcome to the forum.
Line 2: Here you can learn Ruby.
Line 3: Along with other members.
>Exit code: 0
```

2. Run the following two programs and try and understand the difference in the outputs of the two programs. The program:

```
def mtdarry
 10.times do |num|
 puts num
 end
end

mtdarry
and the program:
def mtdarry
 10.times do |num|
 puts num
 end
end

puts mtdarry
```

3. Write a method `leap_year`**?** It should accept a year value from the user, check whether it's a leap year, and then return true or false. With the help of this `leap_year?` method calculate and display the number of minutes in a leap year (2000 and 2004) and the number of minutes in a non-leap year (1900 and 2005). Note: a century year, like 1900 and 2000, is a leap year only if it is divisible by 4.

Note: Though we will be covering the following classes much later, this note here is in order - "***Time*** *and* ***Date***/***DateTime*** *use two entirely different mechanisms for keeping track of the passage of time.* ***Time*** *utilizes seconds since the start of 1970 -- standard Unix time tracking.* ***Date***/***DateTime*** *uses keeps tracks of days and fractions of days using Rational, and it's start of time is about the start of the year in 4712 B.C. They are quite different, and while one should not have to write one's own code to convert from one to another -- all three classes should be patched to allow each conversion from one to another -- they should stay separate. If you want to represent 376 AD, you can't use* ***Time****.* ***Time*** *will be faster, since* ***DateTime*** *uses Rational for everything.* ***DateTime****'s units are days and fractions of days. That might make a difference to you, or it might not. If you are doing a lot of date conversion or date math, you might find* ***DateTime*** *easier to use. Personally, I use* ***Time*** *unless I am dealing with date ranges that go beyond what* ***Time*** *handles, as a general rule.*"

6. This is a sample question from the "Ruby Association Certified Ruby Programmer" examination.
http://www.prometric.com/Ruby/default.htm

Thanks to Satoshi Asakawa for the Japanese to English translation.

Select all correct answers which the following program outputs.

```
title = 'Programming Ruby'
price = 3_990
puts '"#{title}" is #{price} yen.'
```

Answers:

1. "#{title}" is 3_990 yen.
2. "#{title}" is 3990 yen.
3. Programming Ruby is yen.
4. Programming Ruby is #{price} yen.
5. "#{title}" is #{price} yen.
6. Syntax Error

7. 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:

```
puts expand = defined?( expand ) ? expand : true
```

Why is the output nil ?

8. What happens in the following Ruby code?

```
x = 10
y = 20
x, y = y, x
puts x
puts y
```

9. In the following Ruby code, x gets the value **nil**: and pqr remains an undefined local variable. Why?

```
if false
 x = pqr
end
puts x
puts pqr
```

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. In Ruby, the first value in an array has index 0. The **`size`** and **`length`** methods return the number of elements in an array. The last element of the array is at index **`size-1`**. Negative index values count from the end of the array, so the last element of an array can also be accessed with an index of -1. If you attempt to read an element beyond the end of an array (with an index >= **`size`**) or before the beginning of an array (with an index < -**`size`**), Ruby simply returns nil and does not throw an exception. Ruby's arrays are mutable – arrays are dynamically resizable; you can append elements to them and they grow as needed. Let us look at 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]
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]]
# contd. From previous page
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
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** (for any object) 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:

- 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.

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:

```
>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.

Parallel Assignment

To explain this, we'll use the terms lvalue and rvalue. An lvalue is something that can appear on its own on the left-hand side of an

assignment (a variable, constant, or attribute setter method). An rvalue is something that can appear on its own on the right hand side. Ruby lets you have a comma-separated list of rvalues. Once Ruby sees more than one rvalue in an assignment, the rules of parallel assignment come into play. First, all the rvalues evaluated, left to right, and collected into an array (unless they are already an array). This array will be the eventual value returned by the overall assignment. Next, the left hand side (lhs) is inspected. If it contains a single element, the array is assigned to that element.

```
a = 1, 2, 3, 4 # => a == [1, 2, 3, 4]
b = [1, 2, 3, 4] # => b == [1, 2, 3, 4]
```

If the lhs contains a comma, Ruby matches values on the rhs against successive elements on the lhs. Excess elements are discarded.

```
a, b = 1, 2, 3, 4 # => a == 1, b == 2
c, = 1, 2, 3, 4 # => c == 1
```

Environment Variables

An environment variable is a link between our program and the outside world. An environment variable is essentially a label referring to a piece of text; and can be used to store configuration information such as paths, usernames, and so on. You can access operating system environment variables using the predefined variable **ENV**.

```
ENV.each {|k,v| puts "#{k}: #{v}"}
```

Ruby sets **ENV** to the environment variables. After that, iteration proceeds with **each**. This time, the block takes two parameters: k (key) and v (value). Blocks are a completely general mechanism and can take any number of arguments.

The values of some environment variables are read by Ruby when it first starts. These variables modify the behavior of the interpreter, as shown below.

Table 15.1. Environment variables used by Ruby

Variable Name	Description
DLN_LIBRARY_PATH	Search path for dynamically loaded modules.
HOME	Points to user's home directory. Used when expanding ~ in file and directory names.
LOGDIR	Fallback pointer to the user's home directory if $HOME is not set. Used only by Dir.chdir.
OPENSSL_CONF	Specify location of OpenSSL configuration file.
RUBYLIB	Additional search path for Ruby programs ($SAFE must be 0).
RUBYLIB_PREFIX	(Windows only) Mangle the RUBYLIB search path by adding this prefix to each component.
RUBYOPT	Additional command-line options to Ruby; examined after real command-line options are parsed ($SAFE must be 0).
RUBYPATH	With -S option, search path for Ruby programs (defaults to PATH).
RUBYSHELL	Shell to use when spawning a process under Windows; if not set, will also check SHELL or COMSPEC.
RUBY_TCL_DLL	Override default name for TCL shared library or DLL.
RUBY_TK_DLL	Override default name for Tk shared library or DLL. Both this and RUBY_TCL_DLL must be set for either to be used.

A Ruby program may write to the **ENV** object. On most systems this changes the values of the corresponding environment variables. However, this change is local to the process that makes it and to any subsequently spawned child processes. A sub-process changes an environment variable, and this change is inherited by a process that it then starts. However, the change is not visible to the original parent. (This just goes to prove that parents never really know what their children are doing.)

```
ENV["course"] = "FORPC101"
puts "#{ENV['course']}"
```

Command line arguments

If you're starting a program from the command line, you can append parameters onto the end of the command and the program processes them.

You can do the same with your Ruby application. Ruby automatically places any parameters that are appended to the command line when you launch your Ruby program into a special array called **ARGV**. If your program is:

```
f = ARGV[0]
puts f
```

You can execute this program from the command line as:

```
ruby tmp.rb 23
```

The program should display 23.

Library GetoptLong

Class **GetoptLong** supports command-line option parsing. Options may be a minus sign (-) followed by a single character, or two minus signs (--) followed by a name (a long option). Options may be given in any order. A single internal option may have multiple external representations. For example, the option to control verbose output could be any of -v, --verbose, or --details. Some options may also take an associated value. Each internal option is passed to **GetoptLong** as an array, containing strings representing the option's external forms and a flag. The flag specifies how **GetoptLong** is to associate an argument with the option (NO_ARGUMENT, REQUIRED_ARGUMENT, or OPTIONAL_ARGUMENT).

Suppose I want to call a Ruby program as:

```
ruby tsftpc.rb -hftp.ibiblio.org -n21 -uanonymous -
ps@s.com
```

Here's the code to do so:

```
require 'getoptlong'
# Call using "ruby tsftpc.rb -hftp.ibiblio.org -n21 -
uanonymous -ps@s.com"
# The parameters can be in any order
unless ARGV.length == 4
  puts "Usage: ruby tsftpc.rb -hftp_site_url -nport_no -
uuser_name -ppassword"
  exit
end
host_name = port_no = user_name = password = ''
# specify the options we accept and initialize
# the option parser
opts = GetoptLong.new(
[ "--hostname", "-h", GetoptLong::REQUIRED_ARGUMENT ],
[ "--port", "-n", GetoptLong::REQUIRED_ARGUMENT ],
[ "--username", "-u", GetoptLong::REQUIRED_ARGUMENT ],
[ "--pass", "-p", GetoptLong::REQUIRED_ARGUMENT ]
)
# process the parsed options
opts.each do |opt, arg|
  case opt
    when '--hostname'
      host_name = arg
    when '--port'
      port_no = arg
    when '--username'
      user_name = arg
    when '--pass'
      password = arg
  end
end
```

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. We shall be studying **require** in more detail, later on.
http://rubylearning.com/satishtalim/including other files in ruby.html

Also, later on, we shall study how to access constants using **::**

http://rubylearning.com/satishtalim/ruby_constants.html

How do I convert objects into an Array?

If you want to wrap objects in an Array, you can use a special **Kernel** module **Array** method (that starts with a capital letter and looks like a class). This special method converts its argument into an array. For example:

```
str = 'hello'
print Array(str).class # Array
```

Another example:

```
str = 'hello\nworld'
print Array(str) # ["hello\\nworld"]
```

What are the ancestors of Array?

Run the following program, to find that out:

```
a = [1,2,3,4]
print a.class.ancestors
```

You should see:

```
[Array, Enumerable, Object, Kernel, BasicObject]
```

You can refer to all the details of the Array class here:
http://www.ruby-doc.org/ruby-1.9/classes/Array.html

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
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
```

Symbols

A symbol looks like a variable name but it's prefixed with a colon. Examples - **:action**, **:line_items**. You don't have to pre-declare a symbol and they are guaranteed to be unique. There's no need to assign some kind of value to a symbol - Ruby takes care of that for you. Ruby also guarantees that no matter where it appears in your program, a particular symbol will have the same value.

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
# 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

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
    @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
```

Here is another example - **`p039xysymbol.rb`**:

```
# p039xysymbol.rb
know_ruby = :yes
if know_ruby == :yes
  puts 'You are a Rubyist'
else
  puts 'Start learning Ruby'
end
```

The output is:

```
>ruby p039xysymbol.rb
You are a Rubyist
>Exit code: 0
```

In this example, :yes is a symbol. Symbols don't contain values or objects, like variables do. Instead, they're used as a consistent name within code. For example, in the preceding code you could easily replace the symbol with a string, as in example - **p039xyzsymbol.rb**

```
# p039xyzsymbol.rb
know_ruby = 'yes'
if know_ruby == 'yes'
  puts 'You are a Rubyist'
else
  puts 'Start learning Ruby'
end
```

This gives the same result, but isn't as efficient. In this example, every mention of 'yes' creates a new object stored separately in memory, whereas symbols are single reference values that are only initialized once. In the first code example, only :yes exists, whereas in the second example you end up with the full strings of 'yes' and 'yes' taking up memory.

We can also transform a String into a Symbol and vice-versa:

```
puts "string".to_sym.class # Symbol
puts :symbol.to_s.class    # String
```

Symbols are particularly useful when creating hashes and you want to have a distinction between keys and values. Please refer to Using Symbols as Hash Keys for a practical example:
http://rubylearning.com/satishtalim/ruby_hashes.html

Fabio Akita a Brazilian Rails enthusiast, also known online as "AkitaOnRails", wrote this exclusive article on **Ruby Symbols** for the rubylearning.com members like you. Do read the article, after you have gone through this lesson:
http://rubylearning.com/blog/2007/11/26/akitaonrails-on-ruby-symbols/

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** uses hash literals: a list of *key* => *value* pairs between braces.

```
# p040myhash.rb
h = {'dog' => 'canine', 'cat' => 'feline', 'donkey' =>
'asinine', 12 => 'dodecine'}
puts h.length  # 4
puts h['dog']  # 'canine'
puts h
puts h[12]
```

The output is:

```
>ruby p040myhash.rb
4
canine
{"dog"=>"canine", "cat"=>"feline", "donkey"=>"asinine",
12=>"dodecine"}
dodecine
>Exit code: 0
```

Compared with arrays, hashes have one significant advantage: they can use any object as an index.

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: http://railsapi.com/doc/ruby-v1.9/classes/Hash.html

Using Symbols as Hash Keys

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
```

Another example is **p0411symbolhash.rb**

```
# p0411symbolhash.rb
h = {:nickname => 'IndianGuru', :language => 'Marathi',
:lastname => 'Talim'}
puts h
```

The output is:

```
{:nickname=>"IndianGuru", :language=>"Marathi",
:lastname=>"Talim"}
```

Another way of doing the same thing is as shown in **p0412symbolhash.rb**

```
# p0412symbolhash.rb
h = {nickname: 'IndianGuru', language: 'Marathi', lastname:
'Talim'}
puts h
```

The output is:

```
{:nickname=>"IndianGuru", :language=>"Marathi",
:lastname=>"Talim"}
```

Exactly the same as in **p0411symbolhash.rb**

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
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.

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('p014constructs.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 for a full list of modes available.
http://www.ruby-doc.org/ruby-1.9/classes/File.html#M000252
`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. *Always close a file that you open. In the case of a file open for writing, this is very important and can actually prevent lost data.*

`File` implements a **`readlines`** method that reads an entire file into an array, line by line.

Both class methods open and readlines belong to the IO class, whose sub-class is File. We have not done classes, objects, inheritance yet but for the record these two methods are inherited by the sub-class File from the class IO.

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

We shall talk about require soon here:
http://rubylearning.com/satishtalim/including_other_files_in_ruby.html

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
f.close
```

The output is:

```
>ruby p028xrandom.rb
USA'
>Exit code: 0
```

Ruby supports the notion of a file pointer. The file pointer indicates the current location in the file. The **File.new** method opens the file 'hellousa.rb' in *read-only mode* (default mode), returns a new **File** object and the file pointer is positioned at the beginning of the file. In the above program, the next statement is `f.seek(12, IO::SEEK_SET)`. The **seek** method of class **IO**, http://www.ruby-doc.org/ruby-1.9/classes/IO.html
moves the file pointer to a given integer distance (first parameter of **seek** method) in the stream according to the value of the second parameter in the **seek** 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:
http://rubylearning.com/satishtalim/ruby_constants.html

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 reconstitute the object using **Marshal.load**. Ruby uses marshaling to store session data. Refer topic Object Serialization later on:
http://rubylearning.com/satishtalim/object_serialization.html

Some useful Ruby methods - 3

inject

The **inject** method (defined in the module **Enumerable**) lets you accumulate a value across the members of a collection. For example, you can sum all the elements in an array, and find their product, using code such as:

```
puts [1,3,5,7].inject(0) {|sum, element| sum+element} # =>
16
puts [1,3,5,7].inject(1) {|product, element|
product*element} # => 105
```

inject works like this: the first time the associated block is called, sum is set to inject's parameter and element is set to the first element in the collection. The second and subsequent times the block is called, sum is set to the value returned by the block on the previous call. The final value of **inject** is the value returned by the block the last time it was called.

There's one final wrinkle: if **inject** is called with no parameter, it uses the first element of the collection as the initial value and starts the iteration with the second value. This means that we could have written the previous examples as:

```
puts [1,3,5,7].inject {|sum, element| sum+element} # => 16
puts [1,3,5,7].inject {|product, element| product*element}
# => 105
```

clear

arr.clear -> arr

The **Array clear** method removes all elements from arr.

```
a = [ "a", "b", "c", "d", "e" ]
puts a.clear # => []
```

concat

arr.concat(other_array) -> arr

The **Array concat** method appends the elements in other_array to arr.

```
puts [ "a", "b" ].concat( ["c", "d"] ) # => ["a", "b", "c",
"d"]
```

pop

arr.pop -> obj or nil

The **Array pop** method removes the last element from arr and returns it or returns nil if the array is empty.

```
a = [ "a", "m", "z" ]
a.pop # => "z"
puts a # => ["a", "m"]
```

empty?

arr.empty? -> true or false

The **Array empty?** method returns true if arr array contains no elements.

```
puts [].empty? # => true
puts [ 1, 2, 3 ].empty? # => false
```

IO.read(name, [length [, offset]]) -> string

The **IO.read** method opens the file, optionally seeks to the given offset, and then returns length bytes (defaulting to the rest of the file). **read** ensures the file is closed before returning.

A file testfile.txt contains:

```
"This is line one\nThis is line two\nThis is line
three\nAnd so on...\n"
```

Example:

```
IO.read("testfile")

=begin
```

Output is:

```
"This is line one\nThis is line two\nThis is line three\n

And so on...\n"

=end
```

```
IO.read("testfile", 20) # => "This is line one\nThi"

IO.read("testfile", 20, 10) # => "ne one\nThis is line "
```

io.stat -> stat

Returns status information.

```
# Assuming that the file hellousa.rb exists
# new opens the file hellousa.rb in default "r" mode
# and returns a new File object. f =
File.new("hellousa.rb") s = f.stat puts s.atime # => Sat
Jan 26 15:54:52 +0530 2008
```

atime returns the last access time for this file as an object of class **Time**.

file.path -> filename

Returns the pathname used to create *file* as a string. An example:

```
# Assume the following folders exist, then puts
File.new("E:/TalimBackup/rubyprograms/smt.tmp", "w").path
```

displays:

```
E:/TalimBackup/rubyprograms/smt.tmp
```

io.lineno -> int

```
Returns the current line number in io. The stream must be
opened for reading.
```

io.rewind -> 0

Positions io to the beginning of input, resetting lineno to zero.

Create a file called *newtestfile.txt* whose contents are:

```
This is line one\n This is line two\n
```

Run this program:

```
f = File.new("newtestfile.txt") puts f.readline # => "This
is line one\n" f.rewind # => 0 puts f.lineno # => 0 puts
f.readline # => "This is line one\n"
```

The output is:

```
This is line one\n 0 This is line one\n
```

sleep(numeric=0) -> fixnum

The **Kernel sleep** method suspends the current thread for numeric seconds (which may be a Float with fractional seconds). Returns the actual number of seconds slept (rounded), which may be less than that asked for if the thread was interrupted. An argument of zero causes **sleep** to sleep forever.

The program:

```
puts Time.now
sleep 1.9 # roughly 2 seconds puts Time.now
```

gives the output:

```
Thu Jan 31 10:39:48 +0530 2008 Thu Jan 31 10:39:50 +0530
2008
```

Note:
The Enumerable mixin provides collection classes with traversal and searching methods and with the ability to sort. The class must provide a method each, which yields successive members of the collection. Ruby 1.9 adds a substantial number of methods to this module, as well as changing the semantics of many others.

Summary

1. An **Array** is just a list of items in order. 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.

2. Arrays are indexed by integers and the index starts from 0.

3. A trailing comma in an array declaration is ignored.

4. You can access an array beyond its boundary limits; it will return **nil**.

5. We can add more elements to an existing array.

6. Refer to the **Array** documentation for a list of methods.

7. The method **each** (for any object) is an iterator that extracts each element of the array. The method each allows us to do something (whatever we want) to each object the array points to.

8. The variable inside the "goalposts" ie. | | refers to each item in the array as it goes through the loop. You can give this any name you want.

9. 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.

10. The two dot form creates an inclusive range, and the three-dot form creates a range that excludes the specified high value.

11. In Ruby, the sequence 1..100000 is held as a **Range** object containing references to two **Fixnum** objects.

12. The **.to_a** method converts a **Range** to an **Array**.

13. 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.

14. Ranges are not limited to integers or numbers. The beginning and end of a range may be any Ruby object.

15. A symbol looks like a variable name but it's prefixed with a colon.

16. You can think of :id as meaning the name of the variable id, and plain id as meaning the value of the variable.

17. Symbols are useful because a given symbol name refers to the same object throughout a Ruby program.

18. Symbols can be considered constants without values.

19. 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.

20. 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.

21. A **Symbol** object is created by prefixing an operator, string, variable, constant, method, class, module name with a colon.

22. 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.

23. Hashes 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.

24. When you store a value in a hash, you actually supply two objects - the index (normally called the key) and the value.

25. **nil** is returned when an attempt is made to access keys that do not exist in the hash.

26. The method to get a randomly chosen number in Ruby is **rand**.

27. 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.

28. The **File.open** method can open a 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.

29. **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.

30. Always close a file that you open. In the case of a file open for writing, this is very important and can actually prevent lost data.

31. The **seek** method of class **IO**, seeks to a given offset an Integer (first parameter of method) in the stream according to the value of second parameter in the method. The second parameter can be **IO::SEEK_CUR** - Seeks to first integer number parameter plus current position; **IO::SEEK_END** - Seeks to first integer number parameter plus end of stream (you probably want a negative value for first integer number parameter); **IO::SEEK_SET** - Seeks to the absolute location given by first integer number parameter.

Exercise Set 4

1. Why is the output of this program:

```
def method
 a = 50
 puts a
end
a = 10
method
puts a
as shown below?
50
10
```

2. Thanks to Marcos Souza for this exercise. A plain text file has the following contents:

```
test test test test test
test test test test test
test test test test test
test test test test test
test test test test test
test test word test test
test test test test test
test test test test test
test test test test test
test test test test test
test test test test test
```

Observe that in this file, there exists a word 'word'. Write a clever but readable Ruby program that updates this file and the final contents become like this:

```
test test test test test
test test test test test
test test test test test
test test test test test
test test test test test
test test inserted word test test
test test test test test
test test test test test
test test test test test
test test test test test
```

```
test test test test test
```

Do not hard-code the file name.

3. Make use of the class **Dir** for the following -
a. Display your current working directory.
b. Create a new directory *tmp* under your working directory.
c. Change your working directory to *tmp*.
d. Display your current working directory.
e. Go back to your original directory.
f. Delete the *tmp* directory.

4. Given the following Ruby code snippet:

```
a = (1930...1951).to_a
puts a[rand(a.size)]
```

When you run this program, which of the following values will **not** be displayed?

a. 1929
b. 1930
c. 1945
d. 1950
e. 1951
f. 1952

Explain why that value will not be displayed.

Also, have a look at the splat operator:

```
a = [*1930...1951] # splat operator
```

http://github.com/mischa/splat

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.

6. 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 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.

For example:
You enter: **Hello Grandma**
Grandma responds: **HUH?! SPEAK UP, SONNY!**
You enter: **HELLO GRANDMA**
Grandma responds: **NO, NOT SINCE 1938!**

From Chris Pine's Book:
http://pine.fm/LearnToProgram/?Chapter=06

7. First of all, I'd like to thank **Peter Cooper** for allowing me to use this exercise.

The application you are going to develop will be a **text analyzer**. You will be working on it this and next week. Your Ruby code will read in text supplied in a separate file, analyze it for various patterns and statistics, and print out the results for the user. It's not a 3D graphical adventure or a fancy Web site, but text processing programs are the bread and butter of systems administration and most application development. They can be vital for parsing log files and user-submitted text on Web sites, and manipulating other textual data. With this application you will be focusing on implementing the features quickly, rather than developing an elaborate object-oriented structure, any documentation, or a testing methodology.

Your text analyzer will provide the following basic statistics:

Character count
Character count (excluding spaces)
Line count
Word count
Sentence count

Paragraph count
Average number of words per sentence
Average number of sentences per paragraph

In the last two cases, the statistics are easily calculated from the word count, sentence count, and paragraph count. That is, once you have the total number of words and the total number of sentences, it becomes a matter of a simple division to work out the average number of words per sentence.

Before you start to code, the first step is to get some test data that your analyzer can process. You can find the text at:
http://rubylearning.com/data/text.txt

Save the file in the same folder as your other Ruby programs and call it **text.txt**. Your application will read from text.txt by default (although you'll make it more dynamic and able to accept other sources of data later on).

Let me outline the basic steps you need to follow:

1. Load in a file containing the text or document you want to analyze.
2. As you load the file line by line, keep a count of how many lines there are (one of your statistics taken care of).
3. Put the text into a string and measure its length to get your character count.
4. Temporarily remove all whitespace and measure the length of the resulting string to get the character count excluding spaces.
5. Split on whitespace to find out how many words there are.
6. Split on full stops (.), '!' and '?' to find out how many sentences there are.
7. Split on double newlines to find out how many paragraphs there are.
8. Perform calculations to work out the averages.

Create a new, blank Ruby source file and save it as **analyzer.rb** in your Ruby folder.

For now, you should write code marked in yellow ie. points 1 to 3.

8. Write a Ruby program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

9. 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 use **join** to make a new string. Write this program.

10. Write a Ruby program that, when given an array: collection = [1, 2, 3, 4, 5] calculates the sum of its elements.

11. Write a Ruby program that, when given an array: collection = [12, 23, 456, 123, 4579] prints each number, and tells you whether it is odd or even.

12. Something to keep your grey cells ticking.

I have a database of all the course participants. **I want to know the number of participants who have not attempted Quiz 1 in my class**. A student writes a clever Ruby program that creates an array of 0's and 1's. 0's indicate that the participant has not attempted the Quiz and the 1's have attempted it.

Use this array quiz:

```
quiz = [0,0,0,1,0,0,1,1,0,1]
```

and write another clever program to solve my problem. That is **display the number of participants who have not attempted Quiz 1.**

The output of your program should be as follow:
The number of participants who did not attempt Quiz 1 is x out of y total participants.

13. Ruby is a DRY language. Ruby helps you keep your code short and concise. DRY stands for: **Don't Repeat Yourself**. Syntactically, it's a very efficient language: you can express the same thing with less lines of code. As we know, computers are fast enough that more lines of code do not slow them down. But what about you? When it comes to debugging and maintaining, the more code you have to deal with, the harder it is to see what it does and find the problems that need fixing. Here's some code:

```
# The long way
record = Hash.new
record[:name] = "Satish"
record[:email] = "mail@satishtalim.com"
record[:phone] = "919371006659"
```

Rewrite, the above code in one line, the DRY (or Ruby) way.

14. The next set of exercises are sample questions from the Important "Ruby Association Certified Ruby Programmer" examination.
http://www.prometric.com/Ruby/default.htm
Thanks to Satoshi Asakawa for the Japanese to English translation.

*Select all answers which return **true**.*

```
h = { "Ruby" => "Matz", "Perl" => "Larry", "Python" =>
"Guido" }
```

Answers:
1. h.member?("Matz")
2. h.member?("Python")
3. h.include?("Guido")
4. h.include?("Ruby")
5. h.has_value?("Larry")
6. h.exists?("Perl")

15. Select all correct outputs for the following program.

```
for i, j in [[1, 2], [3, 4], [5, 6]]
p [i, j]
end
```

Answers:
1. [[1, 2], nil]
[[3, 4], nil]
[[5, 6], nil]

2. [nil, [1, 2]]
[nil, [3, 4]]
[nil, [5, 6]]

3. [1, 2]
[3, 4]
[5, 6]

4. Syntax Error

16. Select all correct ways to do an ascending sort by string length.

```
a = ["Magazine", "Sunday", "Jump"]
```

Answers:
1. a.sort
2. a.sort { |s| s }
3. a.sort { |l, r| l <=> r }
4. a.sort { |l, r| l.length <=> r.length }
5. a.sort_by { |s| s }
6. a.sort_by { |s| s.length }

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.

```
//.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")
# 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.”

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:

```
/[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 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:
```
/\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.

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]
```

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
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,
http://www.regular-expressions.info/ruby.html
for a more detailed coverage on regular expressions.

The above topic has been adapted from the *Ruby for Rails* book.

Writing our own Class

So far, the procedural style of programming (this continues to be used in languages such as C) was used to write our programs. Programming procedurally means you focus on the steps required to complete a task without paying particular attention to how the data is managed.

In the Object-Orientation style, 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 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.

An object 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. An object is an instance of a class.

The following Class Hierarchy is informative:
http://www.cs.mun.ca/~donald/slug/2003-10-16/presentation/img5.html

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** class 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!

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

```
# define class Dog
# p029dog.rb
# define class Dog
class Dog
  def initialize(breed, name)
    # Instance variables
    @breed = breed
    @name = name
  end

  def bark
    puts 'Ruff! Ruff!'
  end

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

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

The output is:

```
>ruby p029dog.rb
The id of obj is 22982920.
Sorry, the object doesn't understand the 'talk' message.
Ruff! Ruff!
I am of Labrador breed and my name is Benzy
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.
http://en.wikipedia.org/wiki/Dynamic_memory_allocation

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

In Rails: 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 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}."
```

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.

You can ask any object of which class it's a member by using its **Object.class** method. In the above program, if we write the statement:

```
d = Dog.new('Alsatian', 'Lassie')
puts d.class.to_s
```

The output is:

```
>ruby p029dog.rb
Dog
>Exit code: 0
```

instance_of? returns true if object is an instance of the given class, as in this example:

```
num = 10
puts(num.instance_of? Fixnum) # output true
```

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.

Examples:

String - 'hello' or "hello"
Symbol - :symbol or :"hello wrold"
Array - [x, y, z]
Hash - {"India" => "IN"}
Range - 3..7 or 3...7

Garbage Collection

The statement:
```
d =nil
```
makes d a **nil** reference, meaning it does not refer to anything. If I now say:
```
d1 = nil
```
then the Dog object is abandoned and eligible for Garbage collection.
http://whytheluckystiff.net/articles/theFullyUpturnedBin.html

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 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.

Here's an example:

Dog#bark – the instance method **bark** in the class **Dog**
Dog.color – the class method **color**, in the class **Dog**
Dog::color – another way to refer to the class method **color**

In writing about Ruby, the *pound notation* (#) is sometimes used to indicate an instance method - for example, we say **`File.chmod`** to denote the class method **`chmod`** of class **`File`**, and **`File#chmod`** to denote the instance method that has the same name. **This notation is not part of Ruby syntax.**

You will learn how to write class methods, later on here:
http://rubylearning.com/satishtalim/ruby_constants.html

Ruby Method Missing

When you send a message to an object, the object executes the first method it finds on its method lookup path
http://phrogz.net/RubyLibs/RubyMethodLookupFlow.pdf
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 non-existent method, an array of the arguments that were passed in the original call and any block passed to the original method.

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.

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

The output is:

```
>ruby p012zmm.rb.rb
There's no method called anything here -- please try again.
>Exit code: 0
```

*You are also responsible for maintaining the **method_missing** signature*. It's possible to write a hook that captures only a missing method's name while ignoring its arguments and associated block.

Ruby Procs

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'
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
  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!
```

Some useful Ruby methods – 4

class
The method **class** of class **Object**, returns the class of obj, now preferred over **Object#type**, as an object's type in Ruby is only loosely tied to that object's class. This method must always be called with an explicit receiver, as **class** is also a reserved word in Ruby.

```
puts 2.class # => Fixnum
puts 'Ruby'.class # => String
puts nil.class # => NilClass
```

Module.constants -> array
The **constants** method returns an array of the names of all constants defined in the system. This list includes the names of all modules and classes.

```
puts Module.constants.sort
```

loop { block }
The **Kernel** method **loop** repeatedly executes the block.

```
i = 0
loop do
  i += 1
  puts i
  break if i > 100
end
```

abort(msg)
The **Kernel abort** method terminates execution immediately with an exit code of 1. The optional String parameter is written to standard error before the program terminates.

```
p 'hello'
abort('Premature abort')
puts 'hi'
```

obj.between?(min, max) -> true or false

The Module **Comparable** has a method **between?** that returns false if obj <=> min is less than zero or if obj <=> max is greater than zero, true otherwise.

```
puts 3.between?(1, 5) # => true
puts 6.between?(1, 5) # => false
puts 'cat'.between?('ant', 'dog') # => true
puts 'gnu'.between?('ant', 'dog') # => false
```

Exercise Set 5

1. Write a class called Dog, that has name as an instance variable and the following methods:

```
bark(), eat(), chase_cat()
```

I shall create the Dog object as follows:

```
d = Dog.new('Leo')
```

2. Write a Rectangle class. I shall use your class as follows:

```
r = Rectangle.new(23.45, 34.67)
puts "Area is = #{r.area}"
puts "Perimeter is = #{r.perimeter}"
```

3. Modify your Deaf Grandma program (written earlier): 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. You must shout BYE three separate times. If you shout BYEBYEBYE or BYE BYE BYE, grandma should pretend not to hear you (and not count it as a BYE).

From Chris Pine's Book:
http://pine.fm/LearnToProgram/?Chapter=06

4. Write a Ruby program to do the following. Take two text files say A and B. The program should swap the contents of A and B. That is after the program is executed, A should contain B's contents and B should contains A's contents.

5. [Difficulty level: MEDIUM] Write a one-line Ruby script that displays on the screen all the files in the current folder as well as everything in all its sub folders, in sorted order. Make use of Dir.glob method:
http://www.ruby-doc.org/core/classes/Dir.html#M002347
as follows:

```
Dir.glob('**/*')
```

Name this program **inventory.rb**. Create an inventory file by typing the following at the command prompt:

```
ruby inventory.rb > old-inventory.txt
```

After a few days, when some files would have been added / deleted from this folder, run the program again like:

```
ruby inventory.rb > new-inventory.txt
```

Now, write another Ruby script that displays on the screen all the files that have been added in this folder since the time the old-inventory.txt was created.

6. Continue with the previously discussed **TextAnalyzer** program. Complete the following steps:

4. Temporarily remove all whitespace and measure the length of the resulting string to get the character count excluding spaces.
5. Split on whitespace to find out how many words there are.
6. Split on full stops (.), '!' and '?' to find out how many sentences there are.
7. Split on double newlines to find out how many paragraphs there are.
8. Perform calculations to work out the averages.

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 (both are global functions defined in **`Kernel`**, but are used like language keywords) 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'
```

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.

`require(string) => true or false`
Ruby tries to load the library named string, returning true if successful. If the filename does not resolve to an absolute path, it will be searched for in the directories listed in **$:**. If the file has the extension ".rb", it is loaded as a source file; if the extension is ".so", ".o", or ".dll", or whatever the default shared library extension is on the current platform, Ruby loads the shared library as a Ruby extension. Otherwise, Ruby tries adding ".rb", ".so", and so on to the name. The name of the loaded feature is added to the array in **$:**.

In Rails: 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 is already Running'
    else
      @engineState = true
      puts 'Engine Idle'
    end
  end
end
```

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

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

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 'p030motorcycle'
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
m.startEngine
puts self.class
puts self
```

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

One more example is program - p031xdognext.rb

```
require 'p029dog'
# 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.

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

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

If you're writing a new method that conceptually belongs in the original class, you can reopen the class and append your method to the class definition. You should only do this if your method is generally useful, and you're sure it won't conflict with a method defined by some library you include in the future. If your method isn't generally useful, or you don't want to take the risk of modifying a class after its initial creation, create a subclass of the original class. The subclass can override its parent's methods, or add new ones. This is safer because the original class, and any code that depended on it, is unaffected.

Inheritance

Inheritance is a relation between two classes. We know that all cats are mammals, and all mammals are animals. The benefit of inheritance is that classes lower down the hierarchy get the features of those higher up, but can also add specific features of their own. If all mammals breathe, then all cats breathe. *In Ruby, a class can only inherit from a single other class*. Some other languages support multiple inheritance - a feature that allows classes to inherit features from multiple classes, but Ruby **doesn't** support this.

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
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.
http://rubylearning.com/satishtalim/ruby_overriding_methods.html

```
class Bird
  def preen
    puts "I am cleaning my feathers."
  end
  def fly
    puts "I am flying."
  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: http://www.rubyist.net/~slagell/ruby/inheritance.html

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

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
```

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. In Ruby 1.9, **Object** is no longer the root of the class hierarchy. A new class named **BasicObject** serves that purpose, and **Object** is a subclass of **BasicObject**. **BasicObject** is a very simple class, with almost no methods of its own. When

you create a class in Ruby 1.9, you still extend **Object** unless you explicitly specify the superclass, and most programmers will never need to use or extend **BasicObject**.

In Ruby, **initialize** is an ordinary method and is inherited like any other.

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

Inheritance and Instance Variables

Consider the code:

```
class Dog
  def initialize(breed)
    @breed = breed
  end
end

class Lab < Dog
  def initialize(breed, name)
    super(breed)
    @name = name
  end

  def to_s
    "(#@breed, #@name)"
  end
end

puts Lab.new("Labrador", "Benzy").to_s
```

The **to_s** method in class **Lab** references **@breed** variable from the superclass **Dog**. This code works as you probably expect it to:

```
puts Lab.new("Labrador", "Benzy").to_s
```

Outputs --> `(Labrador, Benzy)`

Because this code behaves as expected, you may be tempted to say that these variables are inherited. *That is not how Ruby works*.

All Ruby objects have a set of instance variables. These are **not** defined by the objects's class - they are simply created when a value is assigned to them.

Because instance variables are not defined by a class, they are unrelated to subclassing and the inheritance mechanism.

In the above code, **Lab** defines an initialize method that chains to the **initialize** method of its superclass. The chained method assigns values to the variable **@breed**, which makes those variables come into existence for a particular instance of **Lab**.

The reason that instance variables sometimes appear to be inherited is that instance variables are created by the methods that first assign values to them, and those *methods* are often inherited or chained.

Since instance variables have nothing to do with inheritance, it follows that an instance variable used by a subclass cannot "shadow" an instance variable in the superclass. If a subclass uses an instance variable with the same name as a variable used by one of its ancestors, it will overwrite the value of its ancestor's variable.

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

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:

```
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
```

The output is:

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

We shall be talking in depth about **attr_reader** later:
http://rubylearning.com/satishtalim/ruby_access_control.html

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
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. ***Nothing stops you from defining a method twice*, however the new version takes precedence**.

Abstract class

In Ruby, we can define an *abstract* class that invokes certain undefined "abstract" methods, which are left for subclasses to define.

For example:

```
# This class is abstract; it doesn't define hello or name
# No special syntax is required: any class that invokes
methods
# that are intended for a subclass to implement is abstract
class AbstractKlass
  def welcome
    puts "#{hello} #{name}"
  end
end

# A concrete class
class ConcreteKlass < AbstractKlass
  def hello; "Hello"; end
  def name; "Ruby students"; end
end

ConcreteKlass.new.welcome # Displays "Hello Ruby students"
```

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 (if you define a method with the same name twice, the latter method definition prevails as seen in example `p038or.rb` in topic Ruby Overriding Methods).
http://rubylearning.com/satishtalim/ruby_overriding_methods.html
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-right 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
      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.

Some useful Ruby methods – 5

obj.inspect -> string

The **Object** class has an **inspect** method that returns a string containing a human-readable representation of obj. If not overridden, uses the **to_s** method to generate the string.

```
puts [ 1, 2, 3..4, 'five' ].inspect
```

ObjectSpace.each_object(h class_or_mod i) {| obj | block } -> fixnum

The Ruby runtime system needs to keep track of all known objects. This information is made accessible via the **ObjectSpace.each_object** method. It returns the number of objects found. Objects of Fixnums, Symbols, true, false, and nil are never returned.

```
ObjectSpace.each_object do |obj|
printf "%20s: %s\n", obj.class, obj.inspect
end
```

If you specify a class or module as a parameter to each_object, only objects of that type will be returned.

enum.partition {| obj | block } -> [true_array, false_array]

The **partition** method of module **Enumerable**, divides a collection into two parts.

When partition is called and passed a block, it returns two arrays, the first containing the elements of **enum** for which the block evaluates to true, the second containing the rest.

```
nums = [1,2,3,4,5,6,7,8,9]
odd_even = nums.partition {|x| x%2 ==1}
puts odd_even
```

enum.any? h {| obj | block } -> true or false

The quantifier method **any?** of module **Enumerable** makes it easier to test the nature of a collection.

```
nums = [1,2,3,4,5,6,7,8,9,nil]
# Are any of these numbers even?
flag = nums.any? {|x| x%2 ==0}
puts flag # true
```

It passes each element of the collection to the given block. The method returns **true** if the block ever returns a value other than **false** or **nil**. In the absence of a block, this simply tests the truth value of each element. That is, a block **{|x| x}** is added implicitly.

```
flag1 =nums.any?
# list contains at least one true value
# (non-nil or non-false)
puts flag1 # true
```

enum.all? h {| obj | block } -> true or false

The quantifier method **all?** of module **Enumerable** makes it easier to test the nature of a collection.

```
nums = [1,2,3,4,5,6,7,8,9,nil]
# Are all of these numbers even?
flag = nums.all? {|x| x%2 ==0}
puts flag # false
```

It passes each element of the collection to the given block. The method returns **true** if the block never returns **false** or **nil**. In the absence of a block, this simply tests the truth value of each element. That is, a block **{|x| x}** is added implicitly.

```
flag1 =nums.all?
# list contains no falses or nils
puts flag1 # false
```

Summary

1. 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.
2. Many people find regular expressions difficult to use, difficult to read, un-maintainable, and ultimately counterproductive.
3. You may end up using only a modest number of regular expressions in your Ruby and Rails applications.
4. **Becoming a regular expression wizard isn't a prerequisite for Rails programming.**
5. It's advisable to learn at least the basics of how regular expressions work.
6. A regular expression is simply a way of specifying a pattern of characters to be matched in a string.
7. 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.
8. 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 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.
9. 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.
10. Things an object can *do* are called methods.
11. An object is a combination of state and methods that use the state.
12. A class is used to construct an object. A class is a blueprint for an object.
13. More than 30 built-in classes are predefined in the Ruby class hierarchy. The following class hierarchy is important - http://www.cs.mun.ca/%7Edonald/slug/2003-10-16/presentation/img5.html
14. In Ruby, everything from an integer to a string is considered to be an object. And each object has built in 'methods' (Ruby's 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 **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.

15. Ruby integers are objects of class **Fixnum** or **Bignum**. The **Fixnum** and **Bignum** classes represent integers of differing sizes. Both classes descend from **Integer** (and therefore **Numeric**). The floating-point numbers are objects of class **Float**, corresponding to the native architecture's double data type.
16. A new class is defined typically using class Name ... end
17. Classes in Ruby are first-class objects - each is an instance of class **Class**.
18. MUG THIS UP: **Class** is an object, and **Object** is a class." Hal Fulton
19. When a new class is defined say Name, 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** class 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!
20. Objects are created on the heap.
21. In the statement:
    d = Dog.new('Labrador', 'Benzy')
    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
22. 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:
    puts d.methods
    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.

23. Every object in Ruby has a unique id number associated with it that can be found by the method **object_id**.

24. 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.

25. You can ask any object of which class it's a member by using its **Object.class** method.

26. **instance_of?** returns true if object is an instance of the given class.

27. Literal 'Constructors' means you can use special notation, instead of a call to **new**, to create a new object of that class. Look at the example given for String. Symbol, Array, Hash, Range, Regexp

28. **Garbage Collection** (GC): 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.

29. Variables are used to hold references to objects. Variables themselves have no type, nor are they objects themselves.

30. **`method_missing`** gives you a way to intercept unanswerable messages and handle them gracefully.

31. 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**.

32. 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).

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

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

35. Note that you say **require 'filename'**, not **require 'filename.rb'**.

36. 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.

37. The benefit of inheritance is that classes lower down the hierarchy get the features of those higher up, but can also add specific features of their own.

38. In Ruby, a class can only inherit from a single other class.

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

40. Method overriding allows a subclass to provide a specific implementation of a method that is already provided by one of its superclasses. The implementation in the subclass overrides (replaces) the implementation in the superclass.

41. *Nothing stops you from defining a method twice, however* the new version takes precedence.

42. 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.

43. 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.

44. Called with specific arguments - **super(a, b, c)** - it sends exactly those arguments.

45. A Ruby class can have only one method with a given name.

Exercise Set 6

1. Write a class UnpredictableString which is a sub-class of String. This sub-class should have a method called scramble() which randomly rearranges any string as follows:

```
>ruby unpredictablestring.rb
daano.r n sdt a htIsw taikmgy r
>Exit code: 0
# the original string was: "It was a dark and stormy
night."
```

2. This exercise thanks to Kathy Sierra - http://headrush.typepad.com/ Once upon a time in a software shop, two programmers were given the same spec and told to "build it". The Really Annoying Project Manager forced the two coders to compete, by promising that whoever delivers first gets one of those cool Aeron chairs - http://en.wikipedia.org/wiki/Aeron_chair - all the Silicon Valley guys have. **The spec**. *There will be shapes on a GUI, a square, a circle and a triangle. When the user clicks on a shape, the shape will rotate clockwise 360 degrees (ie. all the way around) and play an AIF sound file specific to that particular shape*. Go for it guys and show me your code.

Note: This exercise is about inheritance, method overriding and method overloading not about UI.

3.

Here are the program specifications for a simple game to be played against the computer.

a. This game is played at the command-line.
b. The game starts when the computer selects three consecutive cells in a 7 cell row (from 0 to 6). When that's complete, the game asks for your first guess.
c. Guess the computer's selected cells in the smallest number of guesses. You are given a rating or level, based on how well you perform.
d. At the command line, the user types in a number from 0 to 6. The computer checks if it's one of the selected cells. If it's a hit, increment the no_of_hits variable. In response to your guess, you'll see a result at the command-line: either "Hit", "Miss" or "End".
e. When you have guessed all three cells, the game ends by printing out your

rating (your number of guesses).

Note:
a. Use only the features we have learned so far, in Ruby.
b. You must design the **GameBoard** class.
c. The `testgameboard.rb` program uses your GameBoard class

```
# testgameboard.rb
require 'gameboard'

# track how many guesses the user makes
no_of_guesses = 0

# instantiate a GameBoard object
gb = GameBoard.new

# make a random number for the first cell,
# and use it to make the cell locations array
random_no = rand(5)
# make an array for the location of the 3
# consecutive ints out of a possible 7
locations = [random_no, random_no+1, random_no+2]

# invoke the setter method of the GameBoard
gb.set_locations_cells(locations)

# variable to track if the game is alive
is_alive = true

while is_alive
  puts 'Enter a number: '
  STDOUT.flush
  user_guess = gets.chomp
  # invoke the check_yourself method on
  # the GameBoard object
  result = gb.check_yourself(user_guess)
  no_of_guesses += 1
  if (result == 'kill')
    is_alive = false
    puts "You took #{no_of_guesses} guesses"
  end
end
```

4. The Really Annoying Project Manager (Yes Boss) has made a spec change to exercise 3!

*"There will be an amoeba shape on the screen, with the others. When the user clicks on the amoeba, it will rotate like the others, and play a **.aif** sound file. The other figures like rectangle rotate around the center. I want the amoeba shape to rotate around a point on one end, like a clock hand !"*

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. The new method uses initialize method as a private helper method.*
- **Protected methods** can be invoked only by objects of the defining class and its subclasses. Access is kept within the family. However, usage of **protected** is limited.
- **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.

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
  def m2                # this method is protected
  end
  private
  def m3                # this method is private
  end
end
ca = ClassAccess.new
ca.m1
#ca.m2
#ca.m3
```

If you remove the comments of the last two statements in the above program, you will get an access violation runtime error.

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

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

Here's an example (`p047zclassaccess.rb`) for 'protected' access control:

```
class Person
  def initialize(age)
    @age = age
  end
  def age
    @age
  end
  def compare_age(c)
    if c.age > age
      "The other object's age is bigger."
    else
      "The other object's age is the same or smaller."
    end
  end
  protected :age
end

marcos = Person.new(34)
chris = Person.new(25)
puts chris.compare_age(marcos)
#puts chris.age
```

The output is:

```
>ruby p047zclassaccess.rb
The other object's age is bigger.
>Exit code: 0
```

In the preceding example, we compare one Person instance with another Person instance. The comparison, however, depends on the result of a call to the method age. The object doing the comparing (chris, in the example) has to ask the other object (marcos) to execute its age method. So, age can't be private.

That's where the protected level comes in. With age protected rather than private, chris can ask marcos to execute age, because chris and marcos are both instances of the same class. But if you try to call the age method of a Person object when self is anything other than a Person object, the method will fail.

A protected method is thus like a private method, but with an exemption for cases where the class of self (chris) and the class of the object having the method called on it (marcos) are the same.

Note that if you remove the comment from the last statement in the program ie. when you use age directly, Ruby throws an exception.

In Ruby, **public, private** and **protected** apply only to methods. Instance and class variables are encapsulated and effectively private, and constants are effectively public. There is no way to make an instance variable accessible from outside a class (except by defining an accessor method). And there is no way to define a constant that is inaccessible to outside use.

Overriding private methods

Private methods cannot be invoked from outside the class that defines them. But they are inherited by subclasses. This means that subclasses can invoke them and can override them.

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
end

song = Song.new("Brazil", "Ivete Sangalo")
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", "Ivete Sangalo")
puts song.name
puts song.artist
```

Exercise 5 Problem 2 Revisited

At the time, we had written the Rectangle class as follows:

```
class Rectangle
  def initialize(b, h)
    @b = b
    @h = h
  end

  def area
    @b * @h
  end

  def perimeter
    2.0 * (@b + @h)
  end
end

r = Rectangle.new(23.45, 34.67)
puts 'Area is = ' + r.area.to_s
puts 'Perimeter is = ' + r.perimeter.to_s
```

We shall now, re-write this class using attribute readers, as follows:

```
class Rectangle
  def initialize(width, height)
    @area = width * height
    @perimeter = 2.0 * (width + height)
  end

  attr_reader :area, :perimeter
end

r = Rectangle.new(23.45, 34.67)
puts 'Area is = ' + r.area.to_s
puts 'Perimeter is = ' + r.perimeter.to_s
```

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 - `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
```

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.

All Ruby objects have a set of instance variables. These are not defined by the object's class - they are simply created when a value is assigned to them. Because instance variables are not defined by a class, they are unrelated to subclassing and the inheritance mechanism.

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
```

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"
```

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

We shall talk about self in more details, later:
http://rubylearning.com/satishtalim/ruby_self.html

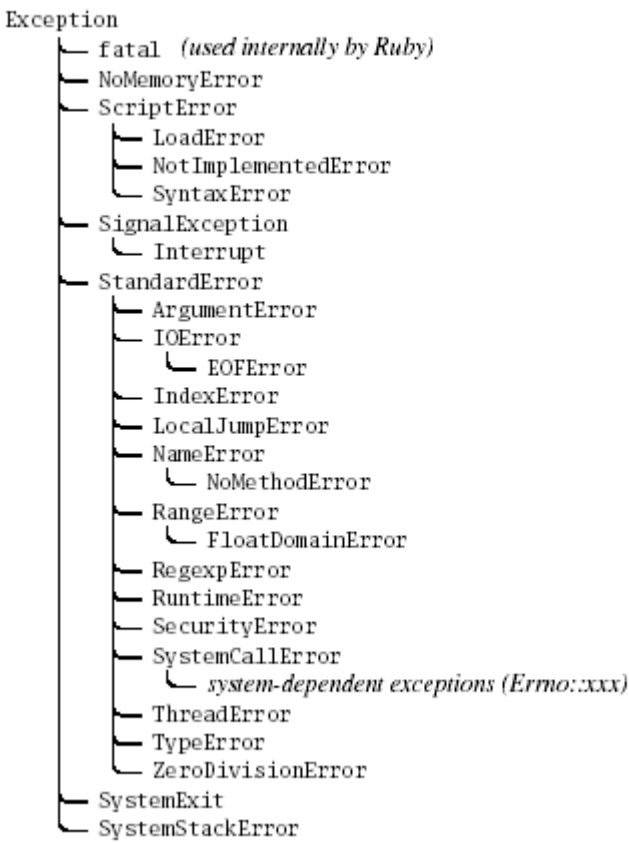
Exceptions

An *exception* is a special kind of object, an instance of the class **Exception** or a descendant of that class that represents some kind of exceptional condition; it indicates that something has gone wrong. When this occurs, an exception is *raised* (or *thrown*). By default, Ruby programs terminate when an exception occurs. But it is possible to declare exception handlers. An exception handler is a block of code that is executed if an exception occurs during the execution of some other block of code. *Raising* an exception means stopping normal execution of the program and transferring the flow-of-control to the exception handling code where you either deal with the problem that's been encountered or exit the program completely. Which of these happens - dealing with it or aborting the program – depends on whether you have provided a **rescue** clause (**rescue** is a fundamental part of the Ruby language). 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 – http://www.ruby-doc.org/core/classes/Exception.html
that help you to handle errors that can occur in your program. The following figure shows the Ruby exception hierarchy.

Figure 8.1. Ruby exception hierarchy



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

The chart above shows that most of the subclasses extend a class known as **StandardError**. These are the "normal" exceptions that typical Ruby programs try to handle. The other exceptions represent lower-level, more serious, or less recoverable conditions, and normal Ruby programs do not typically attempt to handle them.

The following method raises an exception whenever it's called. Its 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.
P043raise.rb:3:in `raise_exception': An error has occured
(RuntimeError)
      from p043raise.rb:6
>Exit code: 1
```

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
P044inverse.rb:2:in `inverse': Argument is not numeric
(ArgumentError)
      from p044inverse.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 **unless** modifier when tacked at the end of a normal statement means execute the preceding expression unless condition is true.

Defining new exception classes
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. It is to be noted that the body of a method definition is an implicit **begin-end** block; the **begin** is omitted, and the entire body of the method is subject to exception handling, ending with the **end** of the method.

The program `p045handexcp.rb` illustrates this.

```
def raise_and_rescue
  begin
    puts 'I am before the raise.'
    raise 'An error has occured.'
    puts 'I am after the raise.'
  rescue
    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 (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
  # …
else
  # Other exceptions
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. The code in an **else** clause is executed if the code in the body of the **begin** statement runs to completion *without* exceptions. If an exception occurs, then the **else** clause will obviously not be executed. The use of an **else** clause is not particularly common in Ruby.

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.
["p046excpvar.rb:2"]
>Exit code: 0
```

The **Exception** class defines two methods that return details about the exception. The **message** method returns a string that may provide human-readable details about what went wrong. The other important method is **backtrace**. This method returns an array of strings that represent the call stack at the point that the exception was raised.

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.

Some common exceptions are:

RuntimeError – this is the default exception raised by the **raise** method.
NoMethodError
NameError
IOError
TypeError
ArgumentError

An Example: Let's modify program `p027readwrite.rb` to include exception handling as shown in example `p046xreadwrite.rb` below.

```
# p046xreadwrite.rb
# Open and read from a text file
# Note that since a block is given, file will automatically
be closed when the block terminates
begin
  File.open('p014constructs.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 two lines of text
    f2.puts "Created by Satish\nThank God!"
  end
rescue Exception => msg
  # display the system generated error message
  puts msg.message
end
```

Improper error messages can provide critical information about an application which may aid an attacker in exploiting the application. The most common problem occurs when detailed internal error messages such as stack traces, database dumps, and error codes are displayed to the user. Security analysts view logging and error handling as potential areas of risk. It is recommended that production applications should not use, for example, a `puts e.backtrace.inspect` call unless it is being directly committed into a log that is not viewable to the end user.

Validation example

Here's an example from the *Ruby Cookbook*, showing how one can do validation of user's inputs.

```
class Name
  # Define default getter methods, but not setter methods.
  attr_reader :first, :last
  # When someone tries to set a first name, enforce rules
about it.
  def first=(first)
    if first == nil or first.size == 0
      raise ArgumentError.new('Everyone must have a first
name.')
    end
    first = first.dup
    first[0] = first[0].chr.capitalize
    @first = first
  end

  # When someone tries to set a last name, enforce rules
about it.
  def last=(last)
    if last == nil or last.size == 0
      raise ArgumentError.new('Everyone must have a last
name.')
    end
    @last = last
  end

  def full_name
    "#{@first} #{@last}"
  end
  # Delegate to the setter methods instead of setting the
instance
  # variables directly.
  def initialize(first, last)
    self.first = first
    self.last = last
  end
end

jacob = Name.new('Jacob', 'Berendes')
jacob.first = 'Mary Sue'
jacob.full_name # => "Mary Sue Berendes"
john = Name.new('john', 'von Neumann')
john.full_name # => "John von Neumann"
john.first = 'john'
john.first # => "John"
```

```
john.first = nil
# ArgumentError: Everyone must have a first name.
Name.new('Kero, international football star and performance
artist', nil)
# ArgumentError: Everyone must have a last name.
```

The Name class keeps track of peoples' first and last names. It uses setter methods to enforce two somewhat parochial rules: everyone must have both a first and a last name, and everyone's first name must begin with a capital letter. The Name class has been written in such a way, that the rules are enforced both in the constructor and after the object has been created. Sometimes you don't trust the data coming in through the setter methods. That's when you can define your own methods to stop bad data before it infects your objects. Within a class, you have direct access to the instance variables. You can simply assign to an instance variable and the setter method won't be triggered. If you do want to trigger the setter method, you'll have to call it explicitly. Note how, in the Name#initialize method above, we call the first= and last= methods instead of assigning to @first and @last. This makes sure the validation code gets run for the initial values of every Name object. We can't just say first = first, because first is a variable name in that method.

Ruby Logging

The **Logger** class in the Ruby standard library – http://www.ruby-doc.org/stdlib/libdoc/logger/rdoc/ 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.

When you write code, you simply assume that all the messages will be logged. At runtime, you can get a more or a less verbose log by changing the log level. A production application usually has a log level of **Logger::INFO** or **Logger::WARN**. From least to most severe, the instance methods are **Logger.debug**, **Logger.info**, **Logger.warn**, **Logger.error**, and **Logger.fatal**.

The DEBUG log level is useful for step-by-step diagnostics of a complex task. The ERROR level is often used when handling exceptions: if the program can't solve a problem, it logs the exception rather than crash and expects a human administrator to deal with it. The FATAL level should only be used when the program cannot recover from a problem, and is about to crash or exit.

If your log is being stored in a file, you can have **Logger** rotate or replace the log file when it gets too big, or once a certain amount of time has elapsed:

```
require 'logger'
# Keep data for the current month only
Logger.new('this_month.log', 'monthly')
# Keep data for today and the past 20 days.
Logger.new('application.log', 20, 'daily')
# Start the log over whenever the log
# exceeds 100 megabytes in size.
Logger.new('application.log', 0, 100 * 1024 * 1024)
```

The code below, uses the application's logger to print a debugging message, and (at a higher severity) as part of error-handling code.

```
#logex.rb
require 'logger'
$LOG = Logger.new('log_file.log', 'monthly')
# $LOG.level = Logger::ERROR
def divide(numerator, denominator)
  $LOG.debug("Numerator: #{numerator}, denominator
#{denominator}")
  begin
    result = numerator / denominator
  rescue Exception => e
    $LOG.error "Error in division!: #{e}"
    result = nil
  end
  return result
end
divide(10, 2)
#divide(10, 0)
```

The contents of the file log_file.log is:

```
# Logfile created on Tue Mar 18 17:09:29 +0530 2008 by /

D, [2008-03-18T17:09:29.216000 #2020] DEBUG -- : Numerator:
10, denominator 2
```

Now try to call the method by:

```
divide(10, 0)
```

The contents of the file log_file.log is:

```
# Logfile created on Tue Mar 18 17:09:29 +0530 2008 by /

D, [2008-03-18T17:09:29.216000 #2020] DEBUG -- : Numerator:
10, denominator 2

D, [2008-03-18T17:13:50.044000 #2820] DEBUG -- : Numerator:
10, denominator 0

E, [2008-03-18T17:13:50.044000 #2820] ERROR -- : Error in
division!: divided by 0
```

To change the log level, simply assign the appropriate constant to level:

```
$LOG.level = Logger::ERROR
```

Now our logger will ignore all log messages except those with severity ERROR or FATAL. The contents of the file log_file.log is:

```
E, [2008-03-18T17:15:59.919000 #2624] ERROR --
 : Error in division!: divided by 0
```

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 – http://www.ruby-doc.org/core/classes/Time.html#M000297
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")

# 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
```

Some useful Ruby methods, keywords, class

obj.send -> other_obj

The **send** method of the **Object** class invokes the method identified by symbol, passing it any arguments and block. You can use **__send__** if the name **send** clashes with an existing method in obj.

Sending messages to objects with the send method:

Here's an example:

```
class Klass
  def hello(*args)
    "Hello " + args.join(' ')
  end
end
k = Klass.new
puts k.send :hello, "gentle", "readers" # => "Hello gentle
readers"
```

Another example is:

Let's say there is a Ticket class as shown below and you don't know which message you want to send?

```
class Ticket
  def venue
    'Town Hall'
  end
  def performer
    'Mark Twain'
  end
  def price
    5.50
  end
  def seat
    'Second Balcony, row J, seat 12'
  end
  def event
  "Author's reading"
  end
end
```

How could that happen? Suppose you want to let a user ”someone at the keyboard” get information from the ticket object. The way you do this (and

yes, there are slicker ways, but bear with me) is to let the user type in the appropriate word ("venue", "performer", and so on) and then display the corresponding value.
Let's start with the keyboard input part. Having created the ticket object and written the methods for it, you'd add this to the program to incorporate keyboard input:

```
print "Information desired: "
request = gets.chomp
```

The code gets a line of keyboard input and chomps off the trailing newline character. At this point, you could proceed as follows, testing the input for one value after another (using the double equal-sign comparison operator (==) and calling the method it matches:

```
if request == "venue"
  puts ticket.venue
elsif request == "performer"
  puts ticket.performer
...
```

You'd continue through the whole list of ticket properties.
Or, you can send the word directly to the ticket object. Instead of the previous code, you do the following:

```
if ticket.respond_to?(request)
  puts ticket.send(request)
else
  puts "No such information available"
end
```

This version uses the **send** method as an all-purpose way of getting a message to the ticket object. It relieves you of having to march through the whole list of possible requests. Instead, having checked that the ticket object will know what to do, you hand the ticket the message and let it do its thing.

Note: It's not uncommon for programs to define a method called send that conflicts with Ruby's built-in **send** method. Therefore, Ruby gives you an alternative way to call send: **__send__**. By convention, no one ever writes a method with that name, so the built-in Ruby version is always available and

never comes into conflict with newly written methods. It looks strange, but it's safer than the plain **send** version from the point of view of method-name clashes.

pp

Ruby comes with a collection of useful Ruby libraries and RubyGems that you might want to use in your programs.

These libraries are documented here - http://www.ruby-doc.org/stdlib/

What you need to know is that

- **bold libraries** in the table of contents are well documented, and
- *italic libraries* are not.

We shall have a look at **pp**.

pp (pretty printer) provides nicer output than a simple **puts something.inspect**. It gives you a nice, clean look at data structures that are properly tabulated and spaced, unlike inspect's output.

The **pp** library is a part of the standard library, so it comes with Ruby by default. To use it, you only need to place this line near the start of your program:

```
require 'pp'
```

Let's look at an example:

```
require 'pp'
class Person
  def initialize(name, sex)
    @name = name
    @sex = sex
  end
end
p1 = Person.new("Peter", "M")
p2 = Person.new("Anita", "F")

people = [p1, p2, p1, p2, p1]
```

The output using puts people.inspect is not so readable as compared to what you would see with pp people.

mod.const_get(symbol) -> obj

symbol refers to a symbol, which is either a quoted string or a Symbol - http://ruby-doc.org/core/classes/Symbol.html
(such as :name - http://ruby-doc.org/core/classes/Module.html#M001699).
Retrieves the value of a constant (by name) from the module or class to which it belongs.

Given a string containing the name of a class, how can we create an instance of that class?

The **Module** method **const_get** can be used to do that. All classes in Ruby are normally named as constants in the "global" namespace - ie. members of **Object**.

```
classname = "String"
klass = Object.const_get(classname)
x = klass.new("Hello")
puts x
```

Struct < Object

Struct is a core Ruby class that generates other classes. These generated classes have accessor methods for the named fields you specify. There are two ways to create a new class with **Struct.new**.

```
Struct.new("Klass", :x, :y) # Creates new class
Struct::Klass
Klass = Struct.new(:x, :y) # Creates new class, assigns to
Klass
```

Naming Anonymous Classes
The second line in the code above, relies on a curious fact about Ruby classes: if you assign an unnamed class object to a constant, the name of the constant becomes the name of a class. You can observe this same behavior if you use the **Class.new** constructor.

```
C = Class.new # A new class with no body, assigned to a
constant
```

```
c = C.new # Create an instance of the class
c.class.to_s # => "C": constant name becomes class name
```

Once a class has been created with **Struct.new**, you can use it like any other class. Its **new** method will expect values for each of the named fields you specify, and its instance methods provide read and write accessors for those fields. You don't have to assign values to all the attributes in the constructor. Those that you omit will be initialized to nil. Here's another example:

```
Course = Struct.new(:name, :city, :country)
participants = Course.new("Satish Talim", "Pune", "India")
puts participants.name
```

undef method-name

The dynamic nature of Ruby means that pretty much anything that can be defined can also be undefined.

To undefine methods, we use the **undef** keyword. Within a class definition, instance methods can be undefined in the same context in which it was defined. You can't **undef** within a method definition. **undef** can also be used to undefine inherited methods, without affecting the definition of the method in the class from which it is inherited. **undef** is not a commonly used statement.

Let's look at an example:

```
def hello
  puts 'hello'
end
hello
undef hello # Any reference to hello now, would give an
error
```

Summary

1. Ruby gives you three levels of protection:
    - **public** - methods can be called by everyone. A class's instance methods are public by default.
    - **protected** - methods can be invoked only by objects of the defining class and its subclasses.
    - **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. The initialize method is always private.
2. 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.
3. Top-level methods are private instance methods of the **Kernel** module.
4. attr_reader is reader only; attr_writer is writer only and attr_accessor is both reader and writer.
5. The **Time** class contains Ruby's interface to the set of time libraries written in C.
6. Time zero for Ruby is the first second GMT of January 1, 1970.
7. Ruby's DateTime class is superior to Time for astronomical and historical applications, but you can use Time for most everyday programs.
8. An exception is a special kind of object, an instance of the class **Exception** or a descendant of that class.
9. 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**.
10. 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.
11. It is to be noted that the body of a method definition is an implicit **begin-end** block; the begin is omitted, and the entire body of the

method is subject to exception handling, ending with the end of the method.

12. If you write a **rescue** clause with no parameter list, the parameter defaults to **StandardError**.
13. 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.
14. By default, the inspect message, which can be sent to any object, formats the object's ID and instance variables. It returns a string containing a human-readable representation of object. If not overridden, uses the to_s method to generate the string.

Exercise Set 7

1. Write a class called Person, that has balance as an instance variable and the following public method: show_balance.
I shall create the Person object as follows:

```
p = Person.new(40000)
puts p.show_bal # calling the method
```

In the above code, 40000 is amount of the initial balance.

2. Write a Ruby program that analyzes an MP3 file. Many MP3 files have a 128-byte data structure at the end called an ID3 tag. These 128 bytes are literally packed with information about the song: its name, the artist, which album it's from, and so on. You can parse this data structure by opening an MP3 file and doing a series of reads from a position near the end of the file. According to the ID3 standard, if you start from the 128th-to-last byte of an MP3 file and read three bytes, you should get the string TAG. If you don't, there's no ID3 tag for this MP3 file, and nothing to do. If there is an ID3 tag present, then the 30 bytes after TAG contain the name of the song, the 30 bytes after that contain the name of the artist, and so on. A sample song.mp3 file is available to test your program –
http://rubylearning.com/data/song.mp3
Use Symbols, wherever possible.

3. Modify your TextAnalyzer program to add the logging feature.

4. Modify your TextAnalyzer program to add the exception feature.

5. Here's code for the part of a game that saves the game state to a file. As a deterrent against cheating, when the game loads a save file it performs a simple check against the file's modification time. If it differs from the timestamp recorded inside the file, the game refuses to load the save file.

The *save_game* method shown below is responsible for recording the timestamp:

```
def save_game(file)
```

```
 score = 1000
 open(file, "w") do |f|
 f.puts(score)
 f.puts(Time.new.to_i)
 end
end
```

The *load_game* method is responsible for comparing the timestamp within the file to the time the filesystem has associated with the file. Write the *load_game(file)* method.

This mechanism can detect simple forms of cheating:

```
save_game("game.sav")
sleep(2)
load_game("game.sav") # => "Your saved score is 1000."
# Now let's cheat by increasing our score to 9000
open("game.sav", "r+b") { |f| f.write("9") }
load_game("game.sav") # RuntimeError: I suspect you of
cheating.
```

Since it's possible to modify a file's times with tools like the Unix touch command, you shouldn't depend on these methods to defend you against a skilled attacker actively trying to fool your program.

Read up on **sleep** method and **Time** class.

Hal Fulton's thoughts on Ruby

a. *Don't write 200 lines of code when 10 will do. A short program fragment will take up less space in the programmer's brain; it will be easier to grasp as a single entity. As a happy side effect, fewer bugs will be injected while the code is being written.* ***Embrace brevity, but do not sacrifice readability****.*

b. *When I took German, I learned that all nouns were capitalized, but the word deutsch was not. I complained to my professor; after all, this was the name of the language, wasn't it? He smiled and said, "Don't fight it." What he taught me was to let German be German. By extension, that is good advice for anyone coming to Ruby from some other language.* ***Let Ruby be Ruby****. Don't expect it to be Perl, because it isn't; don't expect it to be LISP or Smalltalk, either. On the other hand, Ruby has some common elements with all three of these. Start by following your expectations, but when they are violated, don't fight it. (Unless Matz agrees it's a needed change.)*

c. ***Ruby strives to be friendly to the programmer****. For example, there are aliases or synonyms for many method names; size and length will both return the number of entries in an array. The variant spellings indexes and indices both refer to the same method. Some consider this sort of thing to be an annoyance or anti-feature, but I consider it a good design.*

d. ***Ruby strives for consistency and regularity****. For instance, Ruby has the habit of appending a question mark (?) to the name of a predicate like method. This is well and good; it clarifies the code and makes the namespace a little more manageable. But what is more controversial is the similar use of the exclamation point in marking methods that are "destructive" or "dangerous" in the sense that they modify their receivers. The controversy arises because not all of the destructive methods are marked in this way. Shouldn't we be consistent? No, in fact we should not. Some of the methods by their very nature change their receiver (such as the Array methods replace and concat). Some of them are "writer" methods allowing assignment to a class attribute; we should not append an exclamation point to the attribute name or the equal sign. Some methods arguably change the state of the receiver, such as read; this occurs too frequently to be marked in this way. If every destructive method name ended in a !, our programs soon would look like sales brochures for a multilevel marketing firm.* ***What we see in Ruby is not a "foolish consistency" nor a rigid adherence to a set of simple rules****. In language design, as Matz once said, you should "follow your heart."*

e. ***Do not be a slave to performance issues****. When performance is*

unacceptable, the issue must be addressed, but it should normally not be the first thing you think about. Prefer elegance over efficiency where efficiency is less than critical.

Mini Project: Adventure Game (M1 & M2)

Objective: To build a console (non UI), text-based adventure game. Adventure games are full of objects – everything from the locations ('Rooms') to the Treasures they contain. This project is thanks to *Huw Collingbourne*.

Milestone1: The first task is to decide on the main classes which will define the objects in the game. Most game objects – whether they are Rooms, Treasures, Weapons or Monsters – must have at least two properties: a *name* and a *description*. Start by creating a base class, called **Thing**, from which more specialized classes will descend. Write the relevant methods so that the two attributes are accessible outside of the class.

Example usage of the **Thing** class:

```
# t = Thing.new('Satish', 'Ruby Evangelist')
```

Milestone 2: Let's move on to create some more specific *descendent* classes.

a. Class **Room** class is just a **Thing** but it adds on some 'exit' attributes. These attributes will be used to *indicate which room*, if any, is located at the North, South, West and East exits of the current room.

Example usage of the **Room** class:

```
# N S W E
@r0 = Room.new("Treasure Room", "a fabulous golden
chamber", -1, 2, -1, 1)
@r1 = Room.new("Dragon's Lair", "a huge and glittering
lair", -1, -1, 0, -1)
@r2 = Room.new("Troll Cave", "a dank and gloomy cave", 0, -
1, -1, 3)
@r3 = Room.new("Crystal Dome", "a vast dome of glass", -1,
-1, 2, -1)
# 0, 1, 2, 3 above indicate where Rooms @r0…@r3 are located
```

The rooms are located in these positions -

Room 0 Room 1
Room 2 Room 3

Now the code statement:

```
# the numbers indicate positions N S W E
@r0 = Room.new("Treasure Room", "a fabulous golden
chamber", -1, 2, -1, 1)
```

indicates that to the North of Room 0, there is no Room i.e. -1 in the code. To the South of Room 0, there is Room 2 i.e. 2 in the code. To the East of Room 0 is Room 1 i.e. 1 in the code and to the West of Room 0 is no room i.e. -1 in the code.

b. Any adventure game needs a map (a collection of Rooms). For the sake of simplicity, the **Map** class will simply be a class that contains an array of rooms. You could, of course, use the default Ruby **Array** class for this purpose. Or you could make the **Map** class a descendent of **Array**. Let us not use the plain **Array** class for the simple reason that you might want to add special behavior to the **Map** at a later date. Other reasons are:

1. We don't want our code to have access to the whole range of **Array** methods when we use a **Map** object and

2. We may decide to change the **Map** class in a later revision of the code (the Ruby '**Hash**' class – a key/value 'dictionary' – could be used).

Example usage of the **Map** class:

```
# @map = Map.new(someRooms)
```

c. adventure game also needs a Player (to provide the first person perspective as you move through the game).We could create a special one-off class for the **Player**. However, we've decided that we may need more than one object with the ability to move through the game (maybe we'll make it multi-player or maybe we'll add some 'characters' who can move around through the game environment) which is why we've create a more generic class called **Actor**. This class has a position attribute to indicate which room it is in at any given moment.

Example usage of the **Actor** class:

```
# thePlayer = Actor.new("The Player", "You", 0)
```

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.

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.

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

```
class Duck
  def quack
    'Quack!'
  end

  def swim
# contd. From previous page
```

```
    'Paddle paddle paddle...'
  end
end

class Goose
  def honk
    'Honk!'
  end
  def swim
    'Splash splash splash...'
  end
end

class DuckRecording
  def quack
    play
  end

  def play
    'Quack!'
  end
end

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
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.

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
```

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=

```
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')
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.

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

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

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
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.

frozen?

A method **frozen?** tells you whether an object is frozen or not.

```
a = b = 'Original String'
b.freeze
puts a.frozen? # true
puts b.frozen? # true
a = 'New String'
puts a
puts b
puts a.frozen? # false
puts b.frozen? # true
```

Let us understand what we are doing here - a and b are two variables both of which are pointing to a string object - *Original String*. We then freeze the object *Original String*. Hence both a and b are now pointing to the frozen object *Original String*. This is verified by the statements puts a.frozen? and puts b.frozen?. Next, we create a new string object *New String* and make variable a point to this new object *New String*. Variable b is still pointing to the frozen object while a is not. This is verified by the last 2 statements of the program.

Usage:

Ruby sometimes copies objects and freezes the copies. When you use a string as a hash key, Ruby actually copies the string, freezes the copy, and uses the copy as the hash key: that way, if the original string changes later on, the hash key isn't affected.

Ruby's internal file operations work from a frozen copy of a filename instead of using the filename directly. If another thread modifies the original filename in the middle of an operation that's supposed to be atomic, there's no problem: Ruby wasn't relying on the original filename anyway. You can adopt this copy-and-freeze pattern in multi-threaded code to prevent a data structure you're working on from being changed by another thread.

Another common programmer-level use of this feature is to freeze a class in

order to prevent future modifications to it.

Note: Whenever an object in Ruby has no reference to it, then that object is marked for removal and the garbage collector will remove that object based on its algorithm. There is no way to access an un-referenced object.

Summary

1. 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.

2. 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.

3. 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.

4. Syntactic sugar 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.

5. Ruby allows you to define methods that end with an equal sign (=)

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

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

8. 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**.

9. **freeze** operates on an object reference, not on a variable. This means that any operation resulting in a new object will work.

10. A method **frozen?** tells you whether an object is frozen or not.

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.

```
require 'p051gamecharacters'
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'
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
```

Marshal only serializes data structures. It can't serialize Ruby code (like Proc objects), or resources allocated by other processes (like file handles or database connections). Marshal just gives you an error when you try to serialize a file.

Modules/Mixins

Ruby Modules are similar to classes in that they hold a collection of methods, constants, and other module and class definitions. Modules are defined much like classes are, but the **module** keyword is used in place of the **class** keyword. *Unlike classes, you cannot create objects based on modules nor can you subclass them*; 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 stand alone; there is no "module hierarchy" of inheritance. Modules are a good place to collect all your constants in a central location.

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`** illustrate this.

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

```
class EightTrack
  include Debug
  # ...
# contd. From previous page
end

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

Observe how we use **require** or **load**. **require** and **load** take strings as their arguments.

require 'motorcycle' or **load 'motorcycle.rb'**

include takes the name of a module, in the form of a constant, as in **include Stuff**.

The **include** method accepts any number of **Module** objects to mix in: **include Enumerable, Comparable**

Although every class is a module, the **include** method does not allow a class to be included within another class.

Some more examples –

```
#p062stuff.rb
# A module may contain constants, methods and classes.
# No instances
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

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
```

Remember that you can mix in more than one module in a class. However, a class cannot inherit from more than one class. Class names tend to be nouns, while module names are often adjectives.
Class names tend to be nouns, while module names are often adjectives.

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

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 – an instance method of **Object** (even though **self** is not **Object**). Top-level methods are always private.
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. The class of the **main** object is **Object**.

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
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
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:

```
# 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 an 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
```

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

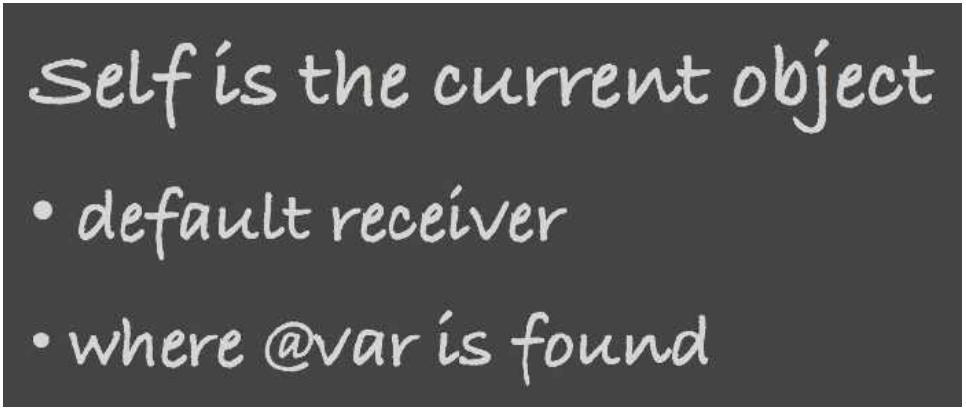
```
# p063xself4.rb

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
```

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

Thus **self** (current object) is the default receiver of method calls and **self** (current object) is where instance variables are found.



Constants

A Ruby constant is like a variable, except that its value is supposed to remain constant for the duration of the program. The Ruby interpreter does not actually enforce the constancy of constants, but it does issue a warning if a program changes the value of a constant (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
```

Lexically, the names of constants look like the names of local variables, except that they begin with a capital letter. By convention, most constants are written in all uppercase with underscores to separate words, LIKE_THIS. Ruby class and module names are also constants, but they are conventionally written using initial capital letters and camel case, LikeThis.

Note that constants do not exist until a value is actually assigned to them.

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
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.

```
# 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
 end
 def self.cla          # class method, getter
   @@cla
 end
 def self.cla=(y)      # class method, setter, also TestVar.
  # contd. From previous page
  @@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
```

Question often asked

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

Looking Ahead

This eBook covers all the Core Ruby features needed for you to get started:

a. with your Ruby on Rails learning process, and / or
b. writing Ruby scripts for System Administration, and / or
c. with learning JRuby, provided you have a Java background, and /or
d. learning Advanced Ruby

At our sister site –
http://rubylearning.org/

you would find relevant courses to help you in your quest to learn Core, Advanced Ruby and JRuby.

Summary

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

2. **Marshal.dump** is used to save a serialized version of an object.

3. **Marshal.load** is used to read in from a serialized object.

4. Ruby 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.

5. Modules serve two purposes: First they act as namespace, letting you define methods whose names will not clash with those defined elsewhere. 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.

6. Observe how we use **require** or **load**. **require** and **load** take strings as their arguments.

   require 'motorcycle' or **load 'motorcycle.rb'**

   include takes the name of a module, in the form of a constant, as in **include 'Stuff'**.

   The **include** method accepts any number of **Module** objects to mix in: **include Enumerable, Comparable**

   Although every class is a module, the **include** method does not allow a class to be included within another class.

7. Remember that you can mix in more than one module in a class. However, a class cannot inherit from more than one class.

8. Class names tend to be nouns, while module names are often adjectives.

9. 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.

10. Please note the rules given for **self** in the Self related page - http://rubylearning.com/satishtalim/ruby_self.html

11. A Ruby constant is a reference to an object.

12. Although constants should not be changed, you can *modify* the internal states of the objects they reference.

13. Remember the rules for constants.

Exercise Set 8

Exercise1. Write a Ruby program named **lesson8exercise1.rb** that defines a class called **Klass** which will be called in another program as follows:

```
obj = Klass.new("hello")
puts obj.say_hello
```

where **say_hello** is a method in that class, which returns the string sent when an object of **Klass** is created. Write another program named **lesson8exercise1a.rb** that creates an object of the class defined in program **lesson8exercise1.rb** and then marshals it and then restores it.

Exercise2. You are using a class (say DTRConvertor) that's got a bug in one of its methods (the bug is that the conversion rate is Rs 38). You know where the bug is and how to fix it, but you can't or don't want to change the source file itself. Write code to do it. The original class is:

```
class DTRConvertor
def convert(dollar_amount)
return dollar_amount * 40.0 # Bug here
end
end
m = DTRConvertor.new
puts m.convert(100.0)
```

Exercise3. Write a method called month_days, that determines the number of days in a month. Usage:

```
days = month_days(5) # 31 (May)
days = month_days(2, 2000) # 29 (February 2000)
```

Remember, you could use the **Date** class here. Read the online documentation for the **Date** class. You must account for leap years in this exercise.

Exercise4. Write a method **last_modified(file)** that takes a file name and displays something like this: file was last modified 125.873605469919 days ago. Use the **Time** class.

Exercise5. Let's say you want to run some Ruby code (such as a call to a shell command) repeatedly at a certain interval (for example, once every five seconds for one minute). Write a method for this. Do not use **Thread** class for now. Hint: **yield** and **sleep** methods may be required.

Mini Project: Adventure Game (M3)

Milestone 3:

Two more classes round things off:

First, a **Game** class which *owns* the **Map.**

Example usage of the **Game** class:

```
@game = Game.new([@r0,@r1,@r2,@r3], @player)
```

Second, an **Implementer** class. The **Implementer** is, in effect, the software equivalent of you – the person who programmed the game. It stands above all other objects and can look down upon and manipulate the entire world of the game with a godlike omniscience. Another way to think of the Implementer is as a sort of chess-player moving pieces (the various objects) around on a board (the map). This means that only one special object, the **Implementer**, owns the game and needs to know where each object is (i.e. in which **Room**) and how to move it from one Room to another.

The **Implementer** starts by initializing the game, then, in response to commands to move the player (or, in principle, any other object of the **Actor** class) in a specific direction, it looks for an exit in the current Room (given by the player's position in the map – **@game.map.rooms[anActor.position]**) and, if it is a positive number, it alters the player's position to the map index given by the new number, otherwise (if the number is -1) there is no exit in that direction and a reply is returned to say so.

Let us test this simple Adventure game by simulating user interaction. In a real game, of course, commands would be entered from the keyboard or using a mouse. For testing purposes, though, we have entered them in code.

```
# Test it out. Create the player and the game
# Then simulate some moves around the map
thePlayer = Actor.new("The Player", "You", 0 )
imp = Implementer.new( thePlayer )
puts( imp.moveActorTo( thePlayer, :e ) )
puts( imp.moveActorTo( thePlayer, :w ) )
puts( imp.moveActorTo( thePlayer, :n ) )
puts( imp.moveActorTo( thePlayer, :s ) )
puts( imp.moveActorTo( thePlayer, :e ) )
puts( imp.moveActorTo( thePlayer, :s ) )
# === The Map ===
```

```
# @r0 -- @r1
# |
# @r2 -- @r3
```

About the Author

Satish Talim (http://satishtalim.com/) is a senior software consultant based in Pune, India with over 30+ 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** (http://www.infonox.com/default.shtml - based in San Jose, CA), **Maybole Technologies Pvt. Ltd.** (http://servient.com/ - Servient Inc. based in Houston, Texas) in Pune, India.

- Has been associated with Java / J2EE since 1995 and involved with Ruby and Ruby on Rails since 2005.

- also started and manages two very active Java and Ruby User Groups in Pune, India – **PuneJava**

  http://tech.groups.yahoo.com/group/pune-java/
  and **PuneRuby**
  http://tech.groups.yahoo.com/group/puneruby/

- Is a Ruby Mentor
  http://rubymentor.rubyforge.org/wiki/wiki.pl?AvailablePureRubyMentors
  on rubyforge.org, helping people with Ruby programming.

He lives in Pune, India, with his wife, son and his Labrador Benzy. In his limited spare time he enjoys travelling and playing online chess.