

An introduction to Ruby

Ruby is commonly regarded as a simple programming language to learn. However it has some features that the average Java programmer has not come across before. This tutorial should help you get started in Ruby.

Installation

If you are running Windows on a machine at home, the simplest way to install Ruby is with the *One-click installer*, available from <http://rubyinstaller.rubyforge.org/>. This gives you a wealth of Ruby goodies, including:

Documentation, with the classic Ruby manual, *Programming Ruby* by Dave Thomas and Andy Hunt (these guys are the Pragmatic Programmers and have published several books on programming - highly recommended). The second edition of *Programming Ruby* is available on desk copy in the library.

The documentation also includes web-links to descriptions of the core API and standard libraries.

RubyGems is a simple but powerful package manager for Ruby applications, but you can safely ignore this at the moment.

There is a nice **editor** that gets installed: **SciTE** – simple and fast text editing with syntax colouring. I recommend editing with SciTE, you can save and run your Ruby file by pressing F5.

You can customize SciTE easily by going to Options:Open Global Options File. One change I recommend is to uncomment (remove the #) the line that says:

```
#buffers = 10
```

This means you can edit more than one file at a time. This might already be done on recent versions.

There is a useful program called **fxri**, which includes quick access to documentation and a version of the Ruby shell, **irb** (interactive ruby). If you are more of the command prompt sort, then you can access the Ruby online documentation with the **ri** command. Type **ri** to get more info on using it.

Last, but not least, there are several directories of Ruby **sample programs**. Browse through them.

If you are running Linux then there is no single installation that provides you with all of the goodies. You certainly need to have Ruby 1.8 installed, and I recommend also installing at least the **ri** and **irb** commands.

First steps

Either go to a command prompt and type **irb** or start up **fxri**. Then type the following instructions:

```
-10.abs
```

This demonstrates that Ruby really is an object-oriented language. The constant value **-10** is sent the message **abs** and the result is the absolute value.

You do not declare variables in Ruby, you just use them. Ruby dynamically associates a class with a variable, depending on what is currently stored in the variable.

```
a = 355; b = 113; c = 12.5; d = "Hi, there!"
```

Ruby can separate statements with a semi-colon, but it is not required if the statements are on different lines.

You can also assign multiple values simultaneously.

```
a, b, c, d = 355, 113, 12.5, "Hi, there!"
```

Has the same effect as the previous line. This means you can do fun things like exchanging the values in two variables like this:

```
a, d = d, a
```

After this, `a` has the value "Hi, there!" and `d` has the value 355.

If you want to know the type of a Ruby object you can call the `class` method.

```
a.class
1.09e-4.class
```

Apart from the standard classes we have used so far (`Fixnum`, `Float` and `String`) there are four other really useful classes in Ruby: `Array`, `Hash`, `Regexp`, and `Range`.

Array

The `Array` class is very powerful and can be used for many collection types. Arrays are similar to Java's `ArrayLists`, but more convenient to use. You can make an empty one in traditional OO style with:

```
my_array = Array.new
```

The same thing can be done using `[]`.

```
my_array = []
```

But we can do much cooler things.

```
my_array = [1, 23.6, "hi", ["another", "array"], 1..10]
```

This array now contains a whole pile of different types of things, including another array, and a range.

See what you get when you type these:

```
my_array.length
my_array[0]
my_array[4]
my_array[3][1]
```

You can even go backwards:

```
my_array[-1]
```

To add some more data into an array you append it with the `<<` method.

```
my_array << "more"
my_array.length
```

In the "Containers, Blocks, and Iterators" chapter of *Programming Ruby* you can see all sorts of ways you can manipulate arrays, in particular inserting and deleting individual elements or groups of elements.

Hash

The `Hash` class provides a quick way to store values and make them retrievable with keys. A `Hash` object can be created in two ways similar to `Array` objects.

```
hash = Hash.new
or
```

```
hash = {}
```

You insert a value into a `Hash` object using its key as the index.

```
hash["hi"] = 16
```

You access the value in the obvious way:

```
hash["hi"]
```

In this case the key is a string and the value is an integer, but objects of any type can be used for both.

There is also a shortcut to define lots of keys and values at the same time.

```
a = {"one" => 1, 2 => "two", :blah => [1, 2, 3, "four"]}
a["one"]
a[2]
a[:blah]
```

Further information on these is in the “Containers, Blocks, and Iterators” chapter of *Programming Ruby*.

Regexp

The Regexp class allows convenient access to regular expressions. Regular expressions can be represented inside a pair of “/”s. You can use other characters to delimit the expression if you need to. These expressions are commonly used to find matches between strings or in case statements, e.g., try the following:

```
"does this MATCH?" =~ /[A-Z]/
```

The =~ is the match operator.

```
/\w+/.match(" find the first word ").to_s
```

The to_s method converts the match result to a string.

```
case "(09)875-1343"
  when /[A-Z, a-z]/
    puts "name"
  when /^[A-Z, a-z]+$/
    puts "name"
  when /^[0-9]+$/
    puts "number"
  when /\^(0[0-9]+\)[0-9]+-[0-9]+$/
    puts "phone number"
end
```

Regular expressions do look like nonsense, but they are incredibly powerful for any string matching tasks. This is particularly useful when manipulating files. That is why most scripting languages include convenient access to them.

For more information see the “Regular Expressions” section in the “Standard Types” chapter of *Programming Ruby*.

Range

There are two main forms of the Range type, the inclusive and the exclusive form.

```
3..5
```

is inclusive and means 3, 4, and 5.

```
3...5
```

is exclusive and means 3 and 4.

Ranges (and almost anything else in Ruby) can be converted to arrays using the to_a method.

```
(3..5).to_a
```

The Ruby Range type is very clever. Try this:

```
("one".."two").to_a
```

But don't try:

```
("one".."three").to_a
```

If you have the time you could try:

```
("one".."three").each { |word| puts word }
```

This shows that we can use ranges for looping.

Ruby names

I haven't said anything about names, but Ruby names its components in strict ways. This quote is from the original version of *Programming Ruby*.

Ruby uses a convention to help it distinguish the usage of a name: the first characters of a name indicate how the name is used. Local variables, method parameters, and method names should all start with a lowercase letter or with an underscore. Global variables are prefixed with a dollar sign (\$), while instance variables begin with an "at" sign (@). Class variables start with two "at" signs (@@). Finally, class names, module names, and constants should start with an uppercase letter.

The Ruby interpreter actually uses the first letter of a name to work out what sort of thing is being referred to. This seems particularly strange to a Java programmer, but is very convenient. Apart from these interpreter enforced understandings, there are also some conventions that Ruby programmers have developed. Variable and method names which are made up of multiple words are usually written with underscores between words, e.g., `long_method_name`. Whereas class names are written in "camel case", e.g., `MyBigClass`. In addition, methods which return a true or false result commonly have a "?" on the end of the name.

Control structures

Ruby has the normal range of control structures. For these examples you can either type them one line at a time into `fxri` or `irb`, or you can save them in a file and run them by typing `ruby filename`. Normally, Ruby programs are saved with a `.rb` extension. Adding this extension to the name also means that the `sciTE` editor realises it is working with a Ruby file and does all of its nice syntax colouring amongst other things.

All Ruby control structures finish with the keyword `end`.

Ruby loops

```
for n in 1..10
  puts n
end
```

```
n = 1
while n <= 10
  puts n
  n += 1
end
```

```
n = 1
until n > 10
  puts n
  n += 1
end
```

```
n = 1
loop do
  puts n
  n += 1
  break if n > 10
end
```

Ruby blocks (part 1)

Ruby has an interesting way of packaging up some code and calling it later. This is the Ruby block and it is delimited by either `{, }` or `do, end`. It is common in Ruby to implement loops with blocks. You might say, "So what? Java does that too." But Ruby blocks are substantially different from anything in Java. They can take parameters and they are actually closures (don't worry if you don't know what a closure is).

Parameters are always passed into a block inside `| |`. The examples below show a single parameter, `n` being passed into the block.

Ruby blocks always occur as part of a method call. The method can then call the block; we will see how in *Ruby blocks (part 2)*.

Here are some more Ruby loops, but this time done with blocks.

```
(1..10).each {|n| puts n}
1.upto(10) {|n| puts n}
10.times {|n| puts n+1}
```

The `each`, `upto` and `times` methods produce a number which is passed to the block (`n` in this case). After the block completes, control returns to the method and it produces the next number and passes it to the block, and so on.

Conditionals

We have already seen one case statement example in the section on regular expressions, but the case statement works with any types, not just regular expressions. Ruby also has a traditional `if` statement.

Methods

Methods can either be defined inside or outside classes. Actually defining a method outside a class actually defines it as a method of the top level `Object` instance (but you probably don't need to know that).

N.B. If a method does not have an explicit `return` statement then the method returns the value of the last statement executed.

```
def factorial(n)
  if n == 0
    return 1
  else
    return n * factorial(n-1)
  end
end

puts factorial(100)
```

Here is another way this can be done in Ruby:

```
def factorial(n)
  return 1 if n == 0
  n * factorial(n-1)
end
```

This shows the `if`-modifier form of a statement. The `return 1` is only executed if the condition is true. If there is no explicit `return` statement, the value of the method is the value of the last statement evaluated.

The next version doesn't use recursion.

```
def factorial(n)
  return 1 if n == 0
  result = 1
  for i in 1..n
    result *= i
  end
  return result
end
```

Methods can't be overloaded in Ruby (not to be confused with overridden, which you must have in normal OOP). So you can't have a method with a variable number of parameters? Well, you can actually, but it is done in a different way.

Default values

You can give parameters default values. Then, when the method is called, if there is no corresponding argument in the call, the parameter gets the default value.

```
def sum(a, b, c = 3, d = 7)
  a + b + c + d
end
```

Can be legally called with

```
sum(1, 4)
sum(3, 4, 9, -20)
sum(-3, 83, 12)
```

Automatic packaging and unpackaging

Another convenience is the way Ruby tries to make sense of assignments between single variables and lists. Try these, and inspect the values of `a`, `b`, and `c` after each:

```
a, b, c = [4, 9, "zoo"]
a, b, c = [4, 9, "zoo", 42]
a, b, c = [4, 9]
```

You can also go the other way:

```
*a = 1, 7, "c"
```

In this case `a` is assigned an array consisting of the elements on the right hand side. This can then be used to pass a different number of parameters to a method as an array.

```
def collection(*data)
  p data
end
collection
collection(4)
collection(4, "word")
```

You can use only one `*parameter` in each method header, and it must follow the normal parameters (but before any optional block parameters).

By the way, the `p` method is like `puts` but it uses a different technique to represent the data. In this case, it shows the array more clearly.

Creating procedures on the go

Like many other languages, Ruby can create procedures (methods) as the program runs. There are several ways of doing this, but the currently preferred way is to use the `lambda` method. Once a procedure object has been created, it can be passed around just like any other object.

```
m = lambda { @buffer.nil? }
```

The `lambda` method takes a block and turns it into a procedure. In this example the procedure will test the instance variable `@buffer` to see if it is `nil`, the same as `null` in Java. The way to call a `lambda` procedure is with the `call` method.

So:

```
@buffer = nil
m.call
returns true. And
```

```
@buffer = []
m.call
returns false.
```

You can also pass parameters as in:

```
square = lambda { |x| x * x }
square.call(3)
This returns 9.
```

You can even do the same thing using a String. So the method could be defined from what a user typed in while the program ran. e.g.

```
print 'Type in your function: '
input = gets.chomp! # the chomp! gets rid of the newline character
# the user then types in: |x| x * x
square = eval('lambda {' + input + '}')
square.call(3)
```

This also returns 9. By the way, the “#” character starts a one line comment in Ruby.

The “eval” method interprets and executes its String parameter.

Classes

One major difference between Ruby and Java classes is that Ruby classes don't declare their instance variables. Remember an instance variable is denoted by starting with the “@” character. So the following Java class:

```
/* A simple point class. */
public class Point {

    private int x;
    private int y;

    /* Create a new point. */
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```

    /* The string version of a Point. */
    public String toString() {
        return "("+x+", "+y+")";
    }
}

/* A line class. */
public class Line {

    private Point[] points = new Point[2];

    /* Create a line. */
    public Line(Point start, Point finish) {
        points[0] = start;
        points[1] = finish;
    }

    /* The string version of a line. */
    public String toString() {
        return "line: ( "+points[0]+points[1]+" )";
    }

    public static void main(String[] args){
        Point p1 = new Point(2, 7);
        Point p2 = new Point(-1, 3);
        Line l = new Line(p1, p2);
        System.out.println(l);
    }
}

```

would be represented in Ruby as:

```

# A simple point class.
class Point

    # Create a new point.
    def initialize(x, y)
        @x, @y = x, y
    end

    # The string version of a point.
    def to_s
        "(#{@x}, #{@y})"
    end

end

# A line class.
class Line

    # Create a line.
    def initialize(start, finish)
        @points = [start, finish]
    end
end

```

```

# The string version of a line.
def to_s
  "line:( #{@points} )"
end

end

p1 = Point.new(2, 7)
p2 = Point.new(-1, 3)
l = Line.new(p1, p2)
puts l

```

I will explain the funny `#{@blah}` strings soon.

N.B. The Ruby classes can all be declared in the same file. Even more importantly the Ruby version can represent points of any type, not just integers.

The `initialize` method basically does the job of a constructor in Java, and is called as a consequence of a call to the `new` method for the class. You subclass another class by adding `< superclass` on the class line.

```

class DoubleArray < Array

  def initialize(list)
    super(list)
    self.collect! {|item| item * 2}
  end

end

puts DoubleArray.new([1, 3, 5, 7, "a"])

```

The call to `super` calls the superclass' method of the same name. In this case the `Array` class' `initialize` method.

Rather than subclassing a particular class, you can make changes directly to a class, even the built-in classes. e.g., If you want to add a method to double all of the elements in an array you can add the method directly to the `Array` class.

```

class Array

  def double
    collect! { |x| x * 2 }
  end

end

[1, 2, 3, 4].double

```

Ruby output

You can use C-style formatted output in Ruby using `printf`. (You can also do this in Java 1.5.) However, in many cases you don't need that much control. The `#{some expression}` construct inside a string, evaluates the expression, converts the result to a string and inserts it in place.

In the `Point` and `Line` classes above, there is actually no need for the curly braces, because the expressions are instance variables and Ruby is smart enough to know what to do.

Here are examples of both output types:

```
a = 45
b = "Hi,"
puts "#{b*5} the sine of #{a} is #{Math.sin(a*Math::PI/180)}"
```

```
c = Math.atan(1)*4
printf("%7s %d %7.4f\n", "hello", 10, c)
```

Ruby blocks (part 2)

So how do methods call Ruby blocks? It is really quite trivial, whenever a method executes the `yield` instruction control passes to the block associated with the method call. When the block finishes, control returns to the method. A `yield` can pass parameters back to the block. In the following example, the `pi` method calculates successive digits of π . By using a block, the same code can be called and different things can be done with each digit, e.g., store the digits, display them, or divide them by 2 and then store them.

```
# From the paper "An Unbounded Spigot Algorithm for the Digits of Pi"
# http://web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/spigot.pdf
```

```
def pi(digits)
  q, r, t, k = 1, 0, 1, 1
  until digits <= 0
    n = (3*q+r) / t
    if (4*q+r) / t == n then
      yield n
      digits -= 1
      q, r, t, k = 10*q, 10*(r-n*t), t, k
    else
      q, r, t, k = q*k, q*(4*k+2)+r*(2*k+1), t*(2*k+1), k+1
    end
  end
end
```

```
stored_pi = []
pi(100) {|d| stored_pi << d}
puts stored_pi.inspect
pi(10) {|d| print d}
puts
stored_half_pi = []
pi(6) {|d| stored_half_pi << d/2}
puts stored_half_pi
```

Sometimes we don't want to call the block directly from the original method, with `yield`. In this case the block can be stored in a variable and called later. e.g.,

```
def store(&block)
  @stored_method = block
end
```

```
store { |x| x + 2 }
@stored_method.call(3)
returns 5.
```

Using an `&` before the last parameter means that it is to be assigned the block object. If there is no block associated with the method call it is given the value `nil`.

Ruby threads

A block can also be executed as a thread. In the following example I use the `do, end` version to delimit the block.

```
million = Thread.new do
  sum = 0
  for i in 1..1000000 do
    sum += i
  end
  sum
end
```

```
$stdout.sync = true # so we see output immediately
puts "waiting for the answer"
answer = million.value
puts "the answer was #{answer}"
```

The thread starts running as soon as it is created.

The next example shows how to pass values into a thread.

```
def simpleThreadProcedure(t)
  for count in (1..100)
    printf "from thread %2d: %3d\n", t, count
  end
end
```

```
threads = []
```

```
for i in (1..10)
  threads << Thread.new(i) do |thread|
    simpleThreadProcedure(thread)
  end
end
```

```
threads.each {|thread| thread.join}
```

The `join` method is like the `value` method, the calling thread waits until the joined thread has completed. The `value` method returns the value of the last statement executed in the thread. The last line above has the main thread wait for all of the threads it has started. If the main thread finishes all other threads die with it.

Thread Variables

Any local variables first used in the thread block, are local to the thread and cannot be accessed by other threads. Sometimes it is necessary to have thread variables accessible to other threads. This is done by treating the Thread object as a Hash, e.g.,

```
t = Thread.new do
  Thread.current["life"] = 42
end
t["life"]
```

I don't recommend using this facility unless absolutely necessary. It is usually better to pass messages between threads.

Unit testing

Like all modern programming languages, there are convenient unit testing libraries for Ruby. If you don't know what unit testing is, then you can't have done many of our courses ☺.

I will give a brief example here.

We want to test the following two methods.

```
# Adds the integers from 0 to n.
# Raises an exception if n is negative.
def add_up_to(n)
  add_range(0, n)
end

# Adds the integers from start to finish.
# Raises an exception if start is > than finish.
def add_range(start, finish)
  raise(ArgumentError, "start > finish", caller) if start > finish
  total = 0
  (start..finish).each { |n| total += n }
  return total
end
```

We normally test it with code in another file, and require 'ourFile' in the test file. But in this case I include the tests directly in the same file.

```
require 'test/unit'

class TestAdder < Test::Unit::TestCase

  def test_adding
    assert_equal(0, add_up_to(0))
    assert_equal(1, add_up_to(1))
    assert_equal(5050, add_up_to(100))
    assert_raise(ArgumentError) { add_up_to(-1) }

    assert_equal(3, add_range(1, 2))
    assert_equal(0, add_range(-100, 100))
    assert_raise(ArgumentError) { add_range(42, 15) }
  end
end
```

end

To use the Unit test framework, you require 'test/unit', subclass Test::Unit::TestCase and start all of your testing methods with the word "test". Then you make assertions, with the expected values or results. In this example two of the assertions specify that the methods in the block should raise exceptions when called.

You run this test file in the usual way. The test framework inspects the test class for methods starting with the word "test" and runs them. The output from this example is:

```
Loaded suite demo
Started
.
Finished in 0.0 seconds.

1 tests, 7 assertions, 0 failures, 0 errors
```

If you want pretty GUI output instead you replace `require 'test/unit'` with `require 'test/unit/ui/tk/testrunner'` and add a line to explicitly call your test class on the end. In this case:

```
Test::Unit::UI::Tk::TestRunner.run(TestAdder)
```

You can find more information on the Test/Unit classes on the local intranet at:

<http://www.cs.auckland.ac.nz/references/ruby/stdlib/libdoc/test/unit/rdoc/classes/Test/Unit.html>.

or on the web at:

<http://www.ruby-doc.org/stdlib/libdoc/test/unit/rdoc/classes/Test/Unit.html>

Ruby gotchas

This section describes some of the confusing things you may come across when reading or writing Ruby code.

What, no parentheses?

You will notice that in many situations the programmers leave parentheses out. You can for example do:

```
def hello_to name
  "Hello, "+ name
end

puts hello_to "you"
```

I strongly recommend that you don't use this style, but you will see it a lot. Always use parentheses. Do as I say, not as I do.

Top-level Ruby

You can program in Ruby ignoring the fact that it is an object-oriented language. E.g.,

```
four = 4

def power_of_4(x)
  x ** 4
end

puts four
puts power_of_4(2)

produces

4
16

but

four = 4

def power_of_4(x)
  x ** four
end

puts four
```

```
puts power_of_4(2)
```

produces

```
4
```

```
NameError: undefined local variable or method `a' for main:Object ...
```

This is because all variables and methods not local to another method or object are added to the base Object.

But constants are different.

```
FOUR = 4
```

```
def power_of_4(x)
  x ** FOUR
end
```

```
puts FOUR
puts power_of_4(2)
```

This works as you might expect - so do instance variables.

```
@four = 4
```

```
def power_of_4(x)
  x ** @four
end
```

```
puts @four
puts power_of_4(2)
```

If you find this confusing, the best thing is to use simple classes always and only refer to obvious constants, local or instance variables, e.g.

```
class Power

  def initialize
    @four = 4
  end

  def power_of_4(x)
    x ** @four
  end

end

p = Power.new
puts p.power_of_4(2)
```

The :: and # notations

When you use `fxri` or `ri` you will see things like, `Thread#value` or `Date::gregorian_leap?(y)`. The `#` means an instance method of the class in Java terms. You don't call it using the `#` symbol; you just use normal dot notation. The `::` is the scope operator. This is used to access class methods and constants (including classes) from outside the class. It can be used instead of normal dot notation as well.

That concludes the crash-course in understanding Ruby. Further Ruby resources are available via the course *Resources* link. Also, feel free to ask me or the tutors for help. The course Wiki should also be a valuable source of information.