Ruby Programming

Ruby is an interpreted, object-oriented scripting language. Its creator, Yukihiro Matsumoto, a.k.a "Matz," released it to the public in 1995. Its history is covered here (http://en.wikipedia.org/wiki/Ruby_Programming_Language) . Its many features are listed here (http://en.wikipedia.org /wiki/Ruby_Programming_Language#Features) .

Ruby was named after the precious gem.

The book is currently broken down into several sections and is intended to be read sequentially. Getting started will show how to install and get started with Ruby in your environment. Basic Ruby demonstrates the main features of the language syntax. The Ruby language section is organized like a reference to the language. Available modules covers some of the standard library. Intermediate Ruby covers a selection of slightly more advanced topics. Each section is designed to be self contained.

Table of Contents

Getting started

Basic Ruby

Ruby Semantic reference

        - `super`
        - Iterators
        - `yield`
    - Classes
        - Class Definition
            - Instance Variables
            - Class Variables
            - Class Methods
            - Instantiation
        - Declaring Visibility
            - Private
            - Public
            - Protected
            - Instance Variables
        - Inheritance
        - Mixing in Modules
        - Ruby Class Meta-Model
        - Ruby Hooks add the ability to know when new methods are defined et al.

See also some rdoc (http://ruby-doc.org/docs/keywords/1.9/) documentation on the various keywords.

Built in Classes

This is a list of classes that are available to you by default in Ruby. They are pre-defined in "core."

- Built-in Functions
- Predefined Variables (globals)
- Predefined Classes
    - `Object`
        - `Array`
        - `Class`
        - `Comparable`
        - `Encoding`
        - `Enumerable`
            - `Enumerable::Enumerator`
        - `Enumerator`
        - `Exception`
        - `FalseClass`
        - `Fiber`
        - `IO`
            - `File`
                - `File::Stat`
            - `GC`
                - `GC::Profiler`
        - `Marshal`
        - `Method`
        - `Math`
        - `Module`
            - `Class`
        - `NilClass`
        - `Numeric`
            - `Integer`
                - `Bignum`
                - `Fixnum`
            - `Float`

- `Range`
- `Regexp`
- `RubyVM`
- `String`
- `Struct`
  - `Struct::Tms`
- `Symbol`
- `Time`
- `Thread`
- `TrueClass`

Available Standard Library Modules

These are parts of Ruby that you have available (in the standard library, or via installation as a gem). To use them you typically have to require some filename, for example `require 'tracer'` would make accessible to you the Tracer class.

You can see a list of basically all the (std lib ruby) modules available in the ruby source (http://github.com/ruby/ruby/tree/trunk/lib/) and lib readme (http://github.com/ruby/ruby/blob/trunk/lib/README) . There are a several more modules available in the std lib, which are C based extensions. You can see their list here (http://github.com/ruby/ruby/tree/trunk/ext) .

- BigDecimal gives you a way to have arbitrary precision Decimal style numbers. Never succumb to rounding errors again!
- Debugger gives you a way to step through debug your Ruby code.
- Distributed Ruby (DRb) gives you a way to make remote procedure calls against objects in a different VM.
- mkmf is a utility used to generate makefiles for ruby extensions.
- Mutex gives you a way to control thread concurrency.
- Net::HTTP gives you a way to download web pages.
- Open3 gives you a way to run a sub-process and have easy access to its I/O.
- OpenSSL is a wrapper to the OpenSSL (C) library, giving you access to secure socket connections.
- Pathname gives you an easy way to manipulate filenames and create/remove files.
- Profiler gives you a way to profile what is taking up the most time in your code.
- OpenURI gives you a way to download files using ruby.
- REXML is a way to parse XML in pure Ruby.
- Ripper gives you a way to parse pure Ruby code into an AST.
- Socket gives you access to Network connectivity.
- Tracer gives you a way to see which lines of your code are being executed and in what order.
- Win32::Registry gives you a way to query and edit the windows registry.
- Win32API gives you a way to call into specific windows core methods easily.
- WIN32OLE gives you a way to use Windows OLE.

Other Libraries

- Database Interface Modules

GUI Libraries

- GUI Toolkit Modules gives a run down of various options for ruby GUI programming.

Here is info on some specifically:

- Tk

- GTK2 Notes on the GTK/Gnome bindings.
- Qt4

Intermediate Ruby

Here are some more in depth tutorials of certain aspects of Ruby.

Unit testing
RubyDoc
Rake
RubyGems
Running Multiple Processes
Using Network Sockets
Building C Extensions
Rails
Embedding Ruby within a separate C program

External links

- Ruby homepage (http://www.ruby-lang.org/)
- Access to various Ruby mailing lists (http://www.ruby-forum.com/)
- Ruby Talk FAQ (http://wiki.github.com/rdp/ruby_tutorials_core/ruby-talk-faq)
- RubyForge: The Repository for Open-source Ruby Projects (http://www.rubyforge.org/)
- [1] (http://rubytrends.com) a place where people vote on things they like most per category

Documentation

Core Docs

- Ruby Documentation Homepage (http://www.ruby-doc.org/)

- ruby-doc.org (http://www.ruby-doc.org) various ruby documentations and tutorials, as well as information on how to update ruby's core docs should you so desire.
- webri (http://webri.tigerops.org) ruby core api as a webri

gem docs

- “rdoc.info”:http://www.rdoc.info yard rdoc’s for gems hosted on github
- “ruby toolbox”:http://ruby-toolbox.com list of gems by popularity

Learning Ruby

- A Ruby Tutorial that Anyone can Edit (http://www.meshplex.org/wiki/Ruby/Ruby_on_Rails_programming_tutorials)
- Learning Ruby (http://www.yoyobrain.com/cardboxes/preview/103) A free tool to find and learn Ruby concepts

Books

Print

- The Ruby Programming Language (http://www.oreilly.com/catalog/9780596516178/) by David

Flanagan, Yukihiro Matsumoto aka “Matz,” the creator of Ruby. Also covers 1.9
- Programming Ruby 3 (http://pragprog.com/titles/ruby3) (aka “Pickaxe”) — this 2009 version covers Ruby 1.9
- Ruby by Example (http://www.oreilly.com/catalog/9781593271480/)

Online

- Programming Ruby (a.k.a. “Pickaxe”) 1st edition online version (http://ruby-doc.org/docs/ProgrammingRuby/)
- Ruby Study Notes (http://rubylearning.com/download/downloads.html)
- Why’s (Poignant) Guide To Ruby (http://www.scribd.com/doc/2236084/Whys-Poignant-Guide-to-Ruby)
- Humble Little Ruby Book (http://www.humblelittlerubybook.com/book/)
- Ruby Hacker’s Guide (http://hawthorne-press.com/WebPage_RHG.html) is a guide to the guts of (mostly 1.8) ruby, translated from its original Japanese.

Quick References

- Ruby Quick Reference (some of more obscure expressions are explained) (http://www.zenspider.com/Languages/Ruby/QuickRef.html)
- Ruby Cheat Sheets (a list of some different Ruby cheat sheets) (http://www.rubyinside.com/ruby-cheat-sheet-734.html)

Retrieved from "http://en.wikibooks.org/w/index.php?title=Ruby_Programming&oldid=2248327"

Ruby Programming/Overview

Ruby is an object-oriented scripting language developed by Yukihiro Matsumoto ("Matz"). The main web site for Ruby is ruby-lang.org (http://www.ruby-lang.org/) . Development began in February 1993 and the first alpha version of Ruby was released in December 1994. It was developed to be an alternative to scripting languages such as Perl and Python.[1] Ruby borrows heavily from Perl and the class library is essentially an object-oriented reorganization of Perl's functionality. Ruby also borrows from Lisp and Smalltalk. While Ruby does not borrow many features from Python, reading the code for Python helped Matz develop Ruby.[1]

Mac OS X comes with Ruby already installed. Most Linux distributions either come with Ruby preinstalled or allow you to easily install Ruby from the distribution's repository of free software. You can also download and install Ruby on Windows. The more technically adept can download the Ruby source code[2] and compile it for most operating systems, including Unix, DOS, BeOS, OS/2, Windows, and Linux.[3]

Features

Ruby combines features from Perl, Smalltalk, Eiffel, Ada, Lisp, and Python.[3]

Object Oriented

Ruby goes to great lengths to be a purely object oriented language. Every value in Ruby is an object, even the most primitive things: strings, numbers and even `true` and `false`. Every object has a *class* and every class has one *superclass*. At the root of the class hierarchy is the class `Object`, from which all other classes inherit.

Every class has a set of *methods* which can be called on objects of that class. Methods are always called on an object — there are no “class methods”, as there are in many other languages (though Ruby does a great job of faking them).

Every object has a set of *instance variables* which hold the state of the object. Instance variables are created and accessed from within methods called on the object. Instance variables are completely private to an object. No other object can see them, not even other objects of the same class, or the class itself. All communication between Ruby objects happens through methods.

Mixins

In addition to classes, Ruby has *modules*. A module has methods, just like a class, but it has no instances. Instead, a module can be included, or “mixed in,” to a class, which adds the methods of that module to the class. This is very much like inheritance but far more flexible because a class can include many different modules. By building individual features into separate modules, functionality can be combined in elaborate ways and code easily reused. Mix-ins help keep Ruby code free of complicated and restrictive class hierarchies.

Dynamic

Ruby is a very *dynamic* programming language. Ruby programs aren’t compiled, in the way that C or Java programs are. All of the class, module and method definitions in a program are built by the code when it is

run. A program can also modify its own definitions while it's running. Even the most primitive classes of the language like String and Integer can be opened up and extended. Rubyists call this *monkey patching* and it's the kind of thing you can't get away with in most other languages.

Variables in Ruby are dynamically typed, which means that any variable can hold any type of object. When you call a method on an object, Ruby looks up the method by name alone — it doesn't care about the type of the object. This is called *duck typing* and it lets you make classes that can pretend to be other classes, just by implementing the same methods.

Singleton Classes

When I said that every Ruby object has a class, I lied. The truth is, every object has *two* classes: a "regular" class and a *singleton class*. An object's singleton class is a nameless class whose only instance is that object. Every object has its very own singleton class, created automatically along with the object. Singleton classes inherit from their object's regular class and are initially empty, but you can open them up and add methods to them, which can then be called on the lone object belonging to them. This is Ruby's secret trick to avoid "class methods" and keep its type system simple and elegant.

Metaprogramming

Ruby is so object oriented that even classes, modules and methods are themselves objects! Every class is an instance of the class `Class` and every module is an instance of the class `Module`. You can call their methods to learn about them or even modify them, while your program is running. That means that you can use Ruby code to generate classes and modules, a technique known as *metaprogramming*. Used wisely, metaprogramming allows you to capture highly abstract design patterns in code and implement them as easily as calling a method.

Flexibility

In Ruby, everything is malleable. Methods can be added to existing classes without subclassing, operators can be overloaded, and even the behavior of the standard library can be redefined at runtime.

Variables and scope

You do not need to declare variables or variable scope in Ruby. The name of the variable automatically determines its scope.

- x is local variable (or something besides a variable).
- $x is a global variable.
- @x is an instance variable.
- @@x is a class variable.

Blocks

Blocks are one of Ruby's most unique and most loved features. A block is a piece of code that can appear after a call to a method, like this:

```
laundry_list.sort do |a,b|
  a.color <=> b.color
end
```

The block is everything between the `do` and the `end`. The code in the block is not evaluated right away, rather it is packaged into an object and passed to the `sort` method as an argument. That object can be called

at any time, just like calling a method. The `sort` method calls the block whenever it needs to compare two values in the list. The block gives you a lot of control over how `sort` behaves. A block object, like any other object, can be stored in a variable, passed along to other methods, or even copied.

Many programming languages support code objects like this. They're called *closures* and they are a very powerful feature in any language, but they are typically underused because the code to create them tends to look ugly and unnatural. A Ruby block is simply a special, clean syntax for the common case of creating a closure and passing it to a method. This simple feature has inspired Rubyists to use closures extensively, in all sorts of creative new ways.

Advanced features

Ruby contains many advanced features.

- Exceptions for error-handling.
- A mark-and-sweep garbage collector instead of reference counting.
- OS-independent threading, which allows you to write multi-threaded applications even on operating systems such as DOS. (this feature will disappear in 1.9, which will use native threads)

You can also write extensions to Ruby in C or embed Ruby in other software.

References

1. ↑ ***a*** ***b*** Bruce Stewart (November 29, 2001). "An Interview with the Creator of Ruby". O'Reilly. http://www.linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html. Retrieved 2006-09-11.
2. ↑ "Download Ruby". http://www.ruby-lang.org/en/downloads/. Retrieved 2006-09-11.
3. ↑ ***a*** ***b*** "About Ruby". http://www.ruby-lang.org/en/about/. Retrieved 2006-09-11.

Ruby Programming/Installing Ruby

Ruby comes preinstalled on Mac OS X and many Linux distributions. In addition, it is available for most other operating systems, including Microsoft Windows.

To find the easiest way to install Ruby for your system, follow the directions below. You can also install Ruby by compiling the source code, which can be downloaded from the Ruby web site (http://www.ruby-lang.org/en/downloads/) .

Operating systems

Mac OS X

Ruby comes preinstalled on Mac OS X. To check what version is on your system:

1. Launch the Terminal application, which is located in the "Utilities" folder, under "Applications".
2. At the command-line, enter: `ruby -v`

If you want to install a more recent version of Ruby, you can:

- Buy the latest version of Mac OS X, which may have a more recent version of Ruby.
- Install Ruby using RVM (http://beginrescueend.com/) . (This is the most popular way because you can manage ruby versions and install many other ruby packages)
- Install Ruby using Fink.
- Install Ruby using MacPorts.
- Install Ruby using Homebrew.

Linux

Ruby comes preinstalled on many Linux systems. To check if Ruby is installed on your system, from the shell run: `ruby -v`

If ruby is not installed, or if you want to upgrade to the latest version, you can usually install Ruby from your distribution's software repository. Directions for some distributions are described below.

Debian / Ubuntu

On Debian and Ubuntu, install Ruby using either the graphical tool Synaptic (on Debian, only if it is installed; it is included with Ubuntu) or the command-line tool apt.

Fedora Core

If you have Fedora Core 5 or later, you can install Ruby using the graphical tool Pirut.[1] Otherwise, you can install Ruby using the command-line tool yum.

Arch Linux

If you have Arch Linux you can install Ruby using the command-line tool pacman.

Mandriva Linux

On Mandriva Linux, install Ruby using the command-line tool urpmi.

PCLinuxOS

On PCLinuxOS, install Ruby using either the graphical tool Synaptic or the command-line tool apt.

Red Hat Linux

On Red Hat Linux, install Ruby using the command-line tool RPM.

Windows

Ruby does not come preinstalled with any version of Microsoft Windows. However, there are several ways to install Ruby on Windows.

- Download and install one of the compiled Ruby binaries from the Ruby web site (http://www.ruby-lang.org/en/downloads/) .
- Download and run the one click RubyInstaller (http://rubyinstaller.org/) .
- Install Cygwin, a collection of free software tools available for Windows. During the install, make sure that you select the "ruby" package, located in the "Devel, Interpreters" category.

Windows is slow

Currently Ruby on windows is a bit slow. Ruby isn't optimized for windows, because most core developers use Linux. Though 1.9.2 passes almost all core tests on windows.

Most of today's slowdown is because when ruby does a

```
require 'xxx'
```

it searches over its entire load path, looking for a file named xxx, or named xxx.rb, or xxx.so or what not. In windows, doing file stat's like that are expensive, so requires take a longer time in windows than linux. 1.9 further complicates the slowdown problem by introducing gem_prelude, which avoids loading full rubygems (a nice speedup actually), but makes the load path larger, so doing require's on windows now takes forever. To avoid this in 1.9.2, you can do a

```
require 'rubygems'
```

which reverts to typical load behavior.

If you want to speed it up (including rails) you can use

http://github.com/rdp/faster_require

Which have some work arounds to make loading faster by caching file locations.

Also the "rubyinstaller" (mingw) builds are faster than the old "one click" installers If yours comes from rubyinstaller.org, chances are you are good there.

NB that Jruby tends to run faster (http://betterlogic.com/roger/?p=2841) but start slower, on windows, than its MRI cousins. Rubinius is currently not yet windows compatible.

Building from Source

If your distro doesn't come with a ruby package or you want to build a specific version of ruby from scratch, please install it by following the directions here (http://svn.ruby-lang.org/repos/ruby/trunk/README) . Download from here (http://www.ruby-lang.org/en/downloads/) .

Compile options

Building with debug symbols

If you want to install it with debug symbols built in (and are using gcc--so either Linux, cygwin, or mingw).

```
./configure --enable-shared optflags="-O0" debugflags="-g3 -ggdb"
```

Optimizations

Note that with 1.9 you can pass it `--disable-install-doc` to have it build faster.

To set the GC to not run as frequently (which tends to provide a faster experience for larger programs, like rdoc and rails), precede your build with

```
$ export CCFLAGS=-DGC_MALLOC_LIMIT=80000000
```

though you might be able to alternately put those in as opt or debug flags, as well.

Testing Installation

The installation can be tested easily.

```
$ ruby -v
```

This should return something like the following:

```
ruby 1.8.7 (2009-06-12 patchlevel 174) [i486-linux]
```

If this shows up, then you have successfully installed Ruby. However, if you get something like the following:

```
-bash: ruby: command not found
```

Then you did not successfully install Ruby.

References

1. ↑ "yum". Fedora Wiki. http://fedoraproject.org/wiki/Tools/yum. Retrieved 2006-09-13.

Ruby Programming/Notation conventions

Command-line examples

In this tutorial, examples that involve running programs on the command-line will use the dollar sign to denote the shell prompt. The part of the example that you type will appear **bold**. Since the dollar sign denotes your shell prompt, you should *not* type it in.

For example, to check what version of Ruby is on your system, run:

```
$ ruby -v
```

Again, do not type the dollar sign – you should only enter "`ruby -v`" (without the quotes). Windows users are probably more familiar seeing "`C:\>`" to denote the shell prompt (called the command prompt on Windows).

An example might also show the output of the program.

```
$ ruby -v
ruby 1.8.5 (2006-08-25) [i386-freebsd4.10]
```

In the above example, "`ruby 1.8.5 (2006-08-25) [i386-freebsd4.10]`" is printed out after you run "`ruby -v`". Your actual output when you run "`ruby -v`" will vary depending on the version of Ruby installed and what operating system you are using.

Running Ruby scripts

For simplicity, the following convention is used to show a Ruby script being run from the shell prompt.

```
$ hello-world.rb
Hello world
```

However, the actual syntax that you will use to run your Ruby scripts will vary depending on your operating system and how it is setup. Please read through the Executable Ruby scripts section of the Hello world page to determine the best way to run Ruby scripts on your system.

Running irb

Ruby typically installs with "interactive ruby" (irb) installed along with it. This is a REPL that allows you to experiment with Ruby, for example:

```
$ irb
>> 3 + 4
=> 7
>> 'abc'
=> "abc"
```

Retrieved from "http://en.wikibooks.org/w/index.php?title=Ruby_Programming/Notation_conventions&

oldid=1845497"

Ruby Programming/Interactive Ruby

When learning Ruby, you will often want to experiment with new features by writing short snippets of code. Instead of writing a lot of small text files, you can use `irb`, which is Ruby's interactive mode.

Running irb

Run `irb` from your shell prompt.

```
$ irb --simple-prompt
>>
```

The `>>` prompt indicates that `irb` is waiting for input. If you do not specify `--simple-prompt`, the `irb` prompt will be longer and include the line number. For example:

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

A simple `irb` session might look like this.

```
$ irb --simple-prompt
>> 2+2
=> 4
>> 5*5*5
=> 125
>> exit
```

These examples show the user's input in **bold**. `irb` uses `=>` to show you the return value of each line of code that you type in.

Cygwin users

If you use Cygwin's Bash shell on Microsoft Windows, but are running the native Windows version of Ruby instead of Cygwin's version of Ruby, read this section.

To run the native version of `irb` inside of Cygwin's Bash shell, run `irb.bat`.

By default, Cygwin's Bash shell runs inside of the Windows console, and the native Windows version of `irb.bat` should work fine. However, if you run a Cygwin shell inside of Cygwin's rxvt terminal emulator, then `irb.bat` will not run properly. You must either run your shell (and `irb.bat`) inside of the Windows console or install and run Cygwin's version of Ruby.

Understanding irb output

`irb` prints out the return value of each line that you enter. In contrast, an actual Ruby program only prints output when you call an output method such as `puts`.

For example:

```
$ irb --simple-prompt
>> x=3
=> 3
>> y=x*2
=> 6
>> z=y/6
=> 1
>> x
=> 3
>> exit
```

Helpfully, `x=3` not only does an assignment, but also returns the value assigned to `x`, which `irb` then prints out. However, this equivalent Ruby program prints nothing out. The variables get set, but the values are never printed out.

```
x=3
y=x*2
z=y/6
x
```

If you want to print out the value of a variable in a Ruby program, use the `puts` method.

```
x=3
puts x
```

Retrieved from "http://en.wikibooks.org/w/index.php?title=Ruby_Programming/Interactive_Ruby&oldid=724569"

Ruby Programming/Hello world

The classic "hello world" program is a good way to get started with Ruby.

Hello world

Create a text file called `hello-world.rb` containing the following code:

```
puts 'Hello world'
```

Now run it at the shell prompt.

```
$ ruby hello-world.rb
Hello world
```

You can also run the short "hello world" program without creating a text file at all. This is called a one-liner.

```
$ ruby -e "puts 'Hello world'"
Hello world
```

You can run this code with irb, but the output will look slightly different. `puts` will print out "`Hello world`", but `irb` will also print out the return value of `puts` – which is `nil`.

```
$ irb --simple-prompt
>> puts "Hello world"
Hello world
=> nil
```

Comments

Like Perl, Bash, and C Shell, Ruby uses the hash symbol (also called Pound Sign, number sign, Square, or octothorpe) for comments. Everything from the hash to the end of the line is ignored when the program is run by Ruby. For example, here's our `hello-world.rb` program with comments.

```
# My first Ruby program
# On my way to Ruby fame & fortune!

puts 'Hello world'
```

You can append a comment to the end of a line of code, as well. Everything before the hash is treated as normal Ruby code.

```
puts 'Hello world'                  # Print out "Hello world"
```

You can also comment several lines at a time:

```
=begin
This program will
print "Hello world".
=end

puts 'Hello world'
```

Although block comments can start on the same line as `=begin`, the `=end` must have its own line. You cannot insert block comments in the middle of a line of code as you can in C, C++, and Java, although you can have non-comment code on the same line as the `=end`.

```
=begin This program will print "Hello world"
=end puts 'Hello world'
```

Executable Ruby scripts

Typing the word `ruby` each time you run a Ruby script is tedious. To avoid doing this, follow the instructions below.

Unix-like operating systems

In Unix-like operating systems – such as Linux, Mac OS X, and Solaris – you will want to mark your Ruby scripts as executable using the `chmod` command. This also works with the Cygwin version of Ruby.

```
$ chmod +x hello-world.rb
```

You need to do this each time you create a new Ruby script. If you rename a Ruby script, or edit an existing script, you do *not* need to run "`chmod +x`" again.

Next, add a shebang line as the *very first line* of your Ruby script. The shebang line is read by the shell to determine what program to use to run the script. This line cannot be preceded by any blank lines or any leading spaces. The new `hello-world.rb` program – with the shebang line – looks like this:

```
#!/usr/bin/ruby

puts 'Hello world'
```

If your `ruby` executable is not in the `/usr/bin` directory, change the shebang line to point to the correct path. The other common place to find the `ruby` executable is `/usr/local/bin/ruby`.

The shebang line is ignored by Ruby – since the line begins with a hash, Ruby treats the line as a comment. Hence, you can still run the Ruby script on operating systems such as Windows whose shell does not support shebang lines.

Now, you can run your Ruby script without typing in the word `ruby`. However, for security reasons, Unix-like operating systems do not search the current directory for executables unless it happens to be listed in your PATH environment variable. So you need to do one of the following:

1. Create your Ruby scripts in a directory that is already in your PATH.
2. Add the current directory to your PATH (*not recommended*).
3. Specify the directory of your script each time you run it.

Most people start with #3. Running an executable Ruby script that is located in the current directory looks like this:

```
$ ./hello-world.rb
```

Once you have completed a script, it's common to create a `~/bin` directory, add this to your PATH, and move your completed script here for running on a day-to-day basis. Then, you can run your script like this:

```
$ hello-world.rb
```

Using env

If you do not want to hard-code the path to the `ruby` executable, you can use the `env` command in the shebang line to search for the `ruby` executable in your PATH and execute it. This way, you will not need to change the shebang line on all of your Ruby scripts if you move them to a computer with Ruby installed in a different directory.

```
#!/usr/bin/env ruby

puts 'Hello world'
```

Windows

If you install the native Windows version of Ruby using the Ruby One-Click Installer (http://www.ruby-lang.org/en/downloads/) , then the installer has setup Windows to automatically recognize your Ruby scripts as executables. Just type the name of the script to run it.

```
$ hello-world.rb
Hello world
```

If this does not work, or if you installed Ruby in some other way, follow these steps.

1. Log in as an administrator.
2. Run the standard Windows "Command Prompt", `cmd`.
3. At the command prompt (*i.e.* shell prompt), run the following Windows commands. When you run `ftype`, change the command-line arguments to correctly point to where you installed the `ruby.exe` executable on your computer.

```
$ assoc .rb=RubyScript
.rb=RubyScript

$ ftype RubyScript="c:\ruby\bin\ruby.exe" "%1" %*
RubyScript="c:\ruby\bin\ruby.exe" "%1" %*
```

For more help with these commands, run "`help assoc`" and "`help ftype`".

Ruby Programming/Strings

Like Python, Java, and the .NET Framework, Ruby has a built-in String class.

String literals

One way to create a String is to use single or double quotes inside a Ruby program to create what is called a string literal. We've already done this with our "hello world" program. A quick update to our code shows the use of both single and double quotes.

```
puts 'Hello world'
puts "Hello world"
```

Being able to use either single or double quotes is similar to Perl, but different from languages such as C and Java, which use double quotes for string literals and single quotes for single characters.

So what difference is there between single quotes and double quotes in Ruby? In the above code, there's no difference. However, consider the following code:

```
puts "Betty's pie shop"
puts 'Betty\'s pie shop'
```

Because "`Betty's`" contains an apostrophe, which is the same character as the single quote, in the second line we need to use a backslash to escape the apostrophe so that Ruby understands that the apostrophe is *in* the string literal instead of marking the end of the string literal. The backslash followed by the single quote is called an escape sequence.

Single quotes

Single quotes only support two escape sequences.

- `\'` – single quote
- `\\` – single backslash

Except for these two escape sequences, everything else between single quotes is treated literally.

Double quotes

Double quotes allow for many more escape sequences than single quotes. They also allow you to embed variables or Ruby code inside of a string literal – this is commonly referred to as interpolation.

```
puts "Enter name"
name = gets.chomp
puts "Your name is #{name}"
```

Escape sequences

Below are some of the more common escape sequences that can appear inside of double quotes.

- `\"` – double quote
- `\\` – single backslash
- `\a` – bell/alert
- `\b` – backspace
- `\r` – carriage return
- `\n` – newline
- `\s` – space
- `\t` – tab

Try out this example code to better understand escape sequences.

```
puts "Hello\t\tworld"

puts "Hello\b\b\b\b\bGoodbye world"

puts "Hello\rStart over world"

puts "1. Hello\n2. World"
```

The result:

```
$ double-quotes.rb
Hello           world
Goodbye world
Start over world
1. Hello
2. World
```

Notice that the newline escape sequence (in the last line of code) simply starts a new line.

The bell character, produced by escape code `\a`, is considered a control character. It does not represent a letter of the alphabet, a punctuation mark, or any other written symbol. Instead, it instructs the terminal emulator (called a console on Microsoft Windows) to "alert" the user. It is up to the terminal emulator to determine the specifics of how to respond, although a beep is fairly standard. Some terminal emulators will flash briefly.

Run the following Ruby code to check out how your terminal emulator handles the bell character.

```
puts "\aHello world\a"
```

puts

We've been using the `puts` function quite a bit to print out text. Whenever `puts` prints out text, it automatically prints out a newline after the text. For example, try the following code.

```
puts "Say", "hello", "to", "the", "world"
```

The result:

```
$ hello-world.rb
Say
hello
to
the
world
```

print

In contrast, Ruby's `print` function only prints out a newline if you specify one. For example, try out the following code. We include a newline at the end of `print`'s argument list so that the shell prompt appears on a new line, after the text.

```
print "Say", "hello", "to", "the", "world", "\n"
```

The result:

```
$ hello-world.rb
Sayhellototheworld
```

The following code produces the same output, with all the words run together.

```
print "Say"
print "hello"
print "to"
print "the"
print "world"
print "\n"
```

See also

String literals (http://en.wikibooks.org/wiki/Ruby_Programming/Syntax/Literals#Strings)

Retrieved from "http://en.wikibooks.org/w/index.php?title=Ruby_Programming/Strings&oldid=2257801"

Ruby Programming/Alternate quotes

In Ruby, there's more than one way to quote a string literal. Much of this will look familiar to Perl programmers.

Alternate single quotes

Let's say we are using single quotes to print out the following path.

```
puts 'c:\bus schedules\napolean\the portland bus schedule.txt'
```

The single quotes keep the `\b`, `\n`, and `\t` from being treated as escape sequences (the same cannot be said for wikibooks' syntax highlighting). But consider the following string literal.

```
puts 'c:\napolean\'s bus schedules\tomorrow\'s bus schedule.txt'
```

Escaping the apostrophes makes the code less readable and makes it less obvious what will print out. Luckily, in Ruby, there's a better way. You can use the `%q` operator to apply single-quoting rules, but choose your own delimiter to mark the beginning and end of the string literal.

```
puts %q!c:\napolean's documents\tomorrow's bus schedule.txt!
puts %q/c:\napolean's documents\tomorrow's bus schedule.txt/
puts %q^c:\napolean's documents\tomorrow's bus schedule.txt^
puts %q(c:\napolean's documents\tomorrow's bus schedule.txt)
puts %q{c:\napolean's documents\tomorrow's bus schedule.txt}
puts %q<c:\napolean's documents\tomorrow's bus schedule.txt>
```

Each line will print out the same text – "`c:\napolean's documents\tomorrow's bus schedule.txt`". You can use any punctuation you want as a delimiter, not just the ones listed in the example.

Of course, if your chosen delimiter appears inside of the string literal, then you need to escape it.

```
puts %q#c:\napolean's documents\tomorrow's \#9 bus schedule.txt#
```

If you use matching braces to delimit the text, however, you can nest braces, without escaping them.

```
puts %q(c:\napolean's documents\the (bus) schedule.txt)
puts %q{c:\napolean's documents\the {bus} schedule.txt}
puts %q<c:\napolean's documents\the <bus> schedule.txt>
```

Alternate double quotes

The `%Q` operator allows you to create a string literal using double-quoting rules, but without using the double quote as a delimiter. It works much the same as the `%q` operator.

```
print %Q^Say:\tHello world\n\tHello world\n^
print %Q(Say:\tHello world\n\tHello world\n)
```

Just like double quotes, you can interpolate Ruby code inside of these string literals.

```
name = 'Charlie Brown'

puts %Q!Say "Hello," #{name}.!
puts %Q/What is "4 plus 5"? Answer: #{4+5}/
```

Retrieved from "http://en.wikibooks.org/w/index.php?title=Ruby_Programming/Alternate_quotes&oldid=2174195"

Ruby Programming/Here documents

For creating multiple-line strings, Ruby supports here documents (heredocs), a feature that originated in the Bourne shell and is also available in Perl and PHP.

Here documents

To construct a here document, the << operator is followed by an identifier that marks the end of the here document. The end mark is called the terminator. The lines of text prior to the terminator are joined together, including the newlines and any other whitespace.

```
puts <<GROCERY_LIST
Grocery list
------------
1. Salad mix.
2. Strawberries.*
3. Cereal.
4. Milk.*

* Organic
GROCERY_LIST
```

The result:

```
$ grocery-list.rb
Grocery list
------------
1. Salad mix.
2. Strawberries.*
3. Cereal.
4. Milk.*

* Organic
```

If we pass the `puts` function multiple arguments, the string literal created from the here document is inserted into the argument list wherever the << operator appears. In the code below, the here-document *(containing the four grocery items and a blank line)* is passed in as the third argument. We get the same output as above.

```
puts 'Grocery list', '------------', <<GROCERY_LIST, '* Organic'
1. Salad mix.
2. Strawberries.*
3. Cereal.
4. Milk.*

GROCERY_LIST
```

You can also have multiple here documents in an argument list. We added a blank line at the end of each here document to make the output more readable.

```
puts 'Produce', '-------', <<PRODUCE, 'Dairy', '-----', <<DAIRY, '* Organic'
1. Strawberries*
2. Blueberries

PRODUCE
1. Yogurt
2. Milk*
3. Cottage Cheese

DAIRY
```

The result:

```
$ grocery-list.rb
Produce
-------
1. Strawberries*
2. Blueberries

Dairy
-----
1. Yogurt
2. Milk*
3. Cottage Cheese

* Organic
```

We have been using the `puts` function in our examples, but you can pass here documents to any function that accepts Strings.

Indenting

If you indent the lines inside the here document, the leading whitespace is preserved. However, there must not be any leading whitepace before the terminator.

```
puts 'Grocery list', '------------', <<GROCERY_LIST
    1. Salad mix.
    2. Strawberries.
    3. Cereal.
    4. Milk.
GROCERY_LIST
```

The result:

```
$ grocery-list.rb
Grocery list
------------
    1. Salad mix.
    2. Strawberries.
    3. Cereal.
    4. Milk.
```

If, for readability, you want to also indent the terminator, use the `<<-` operator.

```
puts 'Grocery list', '------------', <<-GROCERY_LIST
    1. Salad mix.
    2. Strawberries.
    3. Cereal.
    4. Milk.
    GROCERY_LIST
```

Note, however, that the whitespace before each line of text *within* the here document is still preserved.

```
$ grocery-list.rb
Grocery list
------------
    1. Salad mix.
    2. Strawberries.
    3. Cereal.
    4. Milk.
```

Quoting rules

You may wonder whether here documents follow single-quoting or double-quoting rules. If there are no quotes around the identifier, like in our examples so far, then the body of the here document follows double-quoting rules.

```
name = 'Charlie Brown'

puts <<QUIZ
Student: #{name}

1.\tQuestion: What is 4+5?
\tAnswer: The sum of 4 and 5 is #{4+5}
QUIZ
```

The result:

```
$ quiz.rb
Student: Charlie Brown

1.      Question: What is 4+5?
        Answer: The sum of 4 and 5 is 9
```

Double-quoting rules are also followed if you put double quotes around the identifier. However, do not put double quotes around the terminator.

```
puts <<"QUIZ"
Student: #{name}

1.\tQuestion: What is 4+5?
\tAnswer: The sum of 4 and 5 is #{4+5}
QUIZ
```

To create a here document that follows single-quoting rules, place single quotes around the identifier.

```
puts <<'BUS_SCHEDULES'
c:\napolean's documents\tomorrow's bus schedule.txt
c:\new documents\sam spade's bus schedule.txt
c:\bus schedules\the #9 bus schedule.txt
BUS_SCHEDULES
```

The result:

```
$ bus-schedules.rb
c:\napolean's documents\tomorrow's bus schedule.txt
c:\new documents\sam spade's bus schedule.txt
c:\bus schedules\the #9 bus schedule.txt
```

Retrieved from "http://en.wikibooks.org/w/index.php?title=Ruby_Programming/Here_documents&oldid=1898712"

Ruby Programming/ASCII

To us, a string such as "`Hello world`" looks like a series of letters with a space in the middle. To your computer, however, every String – in fact, everything – is a series of numbers.

ASCII

In our example, each character of the String "`Hello world`" is represented by a number between 0 and 127. For example, to the computer, the capital letter "`H`" is encoded as the number 72, whereas the space is encoded as the number 32. The ASCII standard, originally developed for sending telegraphs, specifies what number is used to represent each character.

On most Unix-like operating systems, you can view the entire chart of ASCII codes by typing "`man ascii`" at the shell prompt. Wikipedia's page on ASCII also lists the ASCII codes. Using an ASCII chart, we discover that our string "`Hello world`" gets converted into the following series of ASCII codes.

```
H  e   l   l   o   space  w   o   r   l   d
72 101 108 108 111 32     119 111 114 108 100
```

You can also determine the ASCII code of a character by using the `?` operator in Ruby.

```
puts ?H
puts ?e
puts ?l
puts ?l
puts ?o
```

The question-mark syntax no longer works in Ruby 1.9 (http://stackoverflow.com/questions/1270209/getting-an-ascii-character-code-in-ruby-fails) . Instead, use the **ord** method.

```
puts "H".ord
puts "e".ord
puts "l".ord
puts "l".ord
puts "o".ord
```

Notice that the output (below) of this program matches the ASCII codes for the "`Hello`" part of "`Hello world`".

```
$ hello-ascii.rb
72
101
108
108
111
```

To get the ASCII value for a space, we need to use its escape sequence. In fact, we can use any escape sequence with the `?` operator.

```
puts ?\s
puts ?\t
puts ?\b
```

```
puts ?\a
```

The result:

```
32
9
8
7
```

Terminal emulators

You may not realize it, but so far, you've been running your Ruby programs inside of a program called a terminal emulator – such as the Microsoft Windows console, the Mac OS X Terminal application, a telnet client, rxvt, or X Window System programs such as xterm.

When your Ruby program prints out the letter "H", it sends the ASCII code for "H" (72) to the terminal emulator, which then draws an "H". When your Ruby program prints out a bell character, it sends a different ASCII code – ASCII code 7 – to the terminal emulator. In this case, the terminal emulator does not draw a symbol, but instead will typically beep or flash briefly. How each of the codes gets interpreted is largely determined by the ASCII standard.

Other character encodings

The ASCII standard is a type of character encoding. As mentioned above, ASCII only uses numbers 0 through 127 to define characters. There's a lot more characters than that in the world. Other character encoding systems – such as Latin-1, Shift_JIS, and the Unicode Transformation Format (UTF) – have been created to represent a wider variety of characters, including those found in languages such as Arabic, Hebrew, Chinese, and Japanese.

ASCII chart

Binary	Oct	Dec	Hex	Glyph
010 0000	040	32	20	?
010 0001	041	33	21	!
010 0010	042	34	22	"
010 0011	043	35	23	#
010 0100	044	36	24	$
010 0101	045	37	25	%
010 0110	046	38	26	&
010 0111	047	39	27	'

Binary	Oct	Dec	Hex	Glyph
100 0000	100	64	40	@
100 0001	101	65	41	A
100 0010	102	66	42	B
100 0011	103	67	43	C
100 0100	104	68	44	D
100 0101	105	69	45	E
100 0110	106	70	46	F
100 0111	107	71	47	G

Binary	Oct	Dec	Hex	Glyph
110 0000	140	96	60	`
110 0001	141	97	61	a
110 0010	142	98	62	b
110 0011	143	99	63	c
110 0100	144	100	64	d
110 0101	145	101	65	e
110 0110	146	102	66	f
110 0111	147	103	67	g

010 1000	050	40	28	(
010 1001	051	41	29	)
010 1010	052	42	2A	*
010 1011	053	43	2B	+
010 1100	054	44	2C	,
010 1101	055	45	2D	-
010 1110	056	46	2E	.
010 1111	057	47	2F	/
011 0000	060	48	30	0
011 0001	061	49	31	1
011 0010	062	50	32	2
011 0011	063	51	33	3
011 0100	064	52	34	4
011 0101	065	53	35	5
011 0110	066	54	36	6
011 0111	067	55	37	7
011 1000	070	56	38	8
011 1001	071	57	39	9
011 1010	072	58	3A	:
011 1011	073	59	3B	;
011 1100	074	60	3C	<
011 1101	075	61	3D	=

100 1000	110	72	48	H
100 1001	111	73	49	I
100 1010	112	74	4A	J
100 1011	113	75	4B	K
100 1100	114	76	4C	L
100 1101	115	77	4D	M
100 1110	116	78	4E	N
100 1111	117	79	4F	O
101 0000	120	80	50	P
101 0001	121	81	51	Q
101 0010	122	82	52	R
101 0011	123	83	53	S
101 0100	124	84	54	T
101 0101	125	85	55	U
101 0110	126	86	56	V
101 0111	127	87	57	W
101 1000	130	88	58	X
101 1001	131	89	59	Y
101 1010	132	90	5A	Z
101 1011	133	91	5B	[
101 1100	134	92	5C	\
101 1101	135	93	5D	]

110 1000	150	104	68	h
110 1001	151	105	69	i
110 1010	152	106	6A	j
110 1011	153	107	6B	k
110 1100	154	108	6C	l
110 1101	155	109	6D	m
110 1110	156	110	6E	n
110 1111	157	111	6F	o
111 0000	160	112	70	p
111 0001	161	113	71	q
111 0010	162	114	72	r
111 0011	163	115	73	s
111 0100	164	116	74	t
111 0101	165	117	75	u
111 0110	166	118	76	v
111 0111	167	119	77	w
111 1000	170	120	78	x
111 1001	171	121	79	y
111 1010	172	122	7A	z
111 1011	173	123	7B	{
111 1100	174	124	7C	|
111 1101	175	125	7D	}

011 1110	076	62	3E	>
011 1111	077	63	3F	?

101 1110	136	94	5E	^
101 1111	137	95	5F	_

111 1110	176	126	7E	~

Retrieved from "http://en.wikibooks.org/w/index.php?title=Ruby_Programming/ASCII&oldid=2281905"

Ruby Programming/Introduction to objects

Like Smalltalk, Ruby is a pure object-oriented language — everything is an object. In contrast, languages such as C++ and Java are hybrid languages that divide the world between objects and primitive types. The hybrid approach results in better performance for some applications, but the pure object-oriented approach is more consistent and simpler to use.

What is an object?

Using Smalltalk terminology, an object can do exactly three things.

1. Hold state, including references to other objects.
2. Receive a message, from both itself and other objects.
3. In the course of processing a message, send messages, both to itself and to other objects.

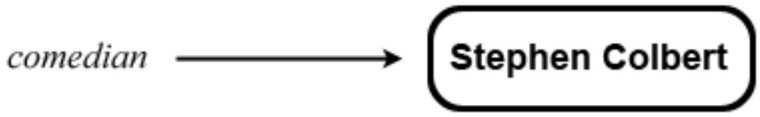
If you don't come from Smalltalk background, it might make more sense to rephrase these rules as follows:

1. An object can contain data, including references to other objects.
2. An object can contain methods, which are functions that have special access to the object's data.
3. An object's methods can call/run other methods/functions.

Variables and objects

Let's fire up `irb` to get a better understanding of objects.

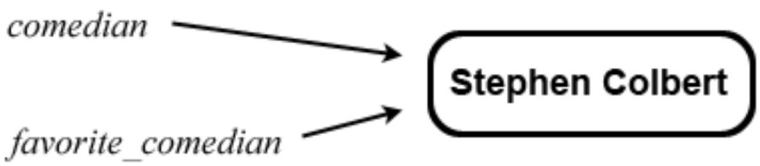
```
$ irb --simple-prompt
>> comedian = "Stephen Colbert"
=> "Stephen Colbert"
```



In the first line, we created a String object containing the text "`Stephen Colbert`". We also told Ruby to use the variable `comedian` to refer to this object.

Next, we tell Ruby to also use the variable `favorite_comedian` to refer to the same String object.

```
>> favorite_comedian = comedian
=> "Stephen Colbert"
```



Now, we have two variables that we can use to refer to the same String object — `comedian` and

`favorite_comedian`. Since they both refer to the same object, if the object changes (as we'll see below), the change will show up when using either variable.

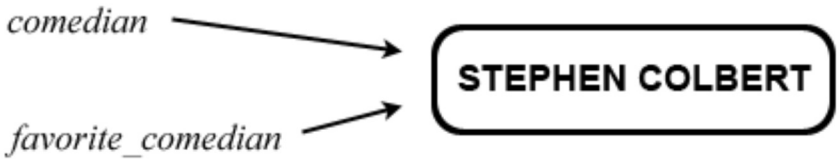
Methods

In Ruby, methods that end with an exclamation mark (also called a "bang") modify the object. For example, the method `upcase!` changes the letters of a String to uppercase.

```
>> comedian.upcase!
=> "STEPHEN COLBERT"
```

Since both of the variables `comedian` and `favorite_comedian` point to the same String object, we can see the new, uppercase text using either variable.

```
>> comedian
=> "STEPHEN COLBERT"
>> favorite_comedian
=> "STEPHEN COLBERT"
```



Methods that do not end in an exclamation point return data, but do not modify the object. For example, `downcase!` modifies a String object by making all of the letters lowercase. However, `downcase` returns a lowercase copy of the String, but the original string remains the same.

```
>> comedian.downcase
=> "stephen colbert"
>> comedian
=> "STEPHEN COLBERT"
```

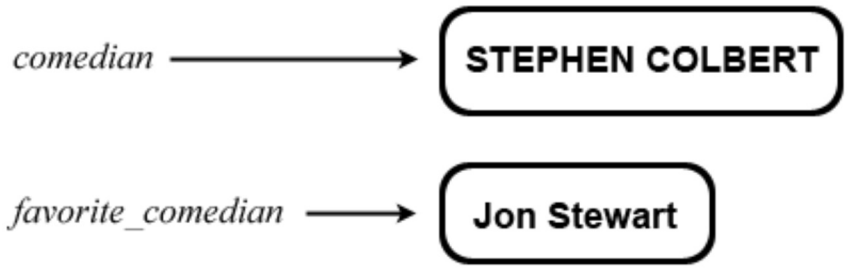
Since the original object still contains the text "`STEPHEN COLBERT`", you might wonder where the new String object, with the lowercase text, went to. Well, after `irb` printed out its contents, it can no longer be accessed since we did not assign a variable to keep track of it.It's essentially gone, and Ruby will dispose of it.

Reassigning a variable

But what if your favorite comedian is not Stephen Colbert? Let's point `favorite_comedian` to a new object.

```
>> favorite_comedian = "Jon Stewart"
=> "Jon Stewart"
```

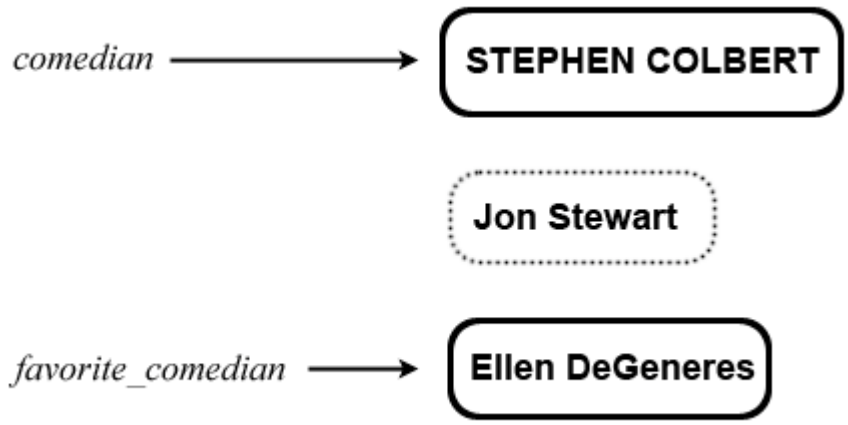
Now, each variable points to a different object.



Let's say that we change our mind again. Now, our favorite comedian is Ellen DeGeneres.

```
>> favorite_comedian = "Ellen DeGeneres"
=> "Ellen DeGeneres"
```

Now, no variable points to the "`Jon Stewart`" String object any longer. Hence, Ruby will dispose of it.



Retrieved from "http://en.wikibooks.org/w/index.php?title=Ruby_Programming/Introduction_to_objects&oldid=854422"

Ruby Programming/Ruby basics

As with the rest of this tutorial, we assume some basic familiarity with programming language concepts (i.e. if statement, while loops) and also some basic understanding of object-oriented programming.

Dealing with variables

We'll deal with variables in much more depth when we talk about classes and objects. For now, let's just say your basic local variable names should start with either a lower case letter or an underscore, and should contain upper or lower case letters, numbers, and underscore characters. Global variables start with a `$`.

Program flow

Ruby includes a pretty standard set of looping and branching constructs: `if`, `while` and `case`

For example, here's `if` in action:

```
a = 10 * rand

if a < 5
  puts "#{a} less than 5"
elsif a > 7
  puts "#{a} greater than 7"
else
  puts "Cheese sandwich!"
end
```

[As in other languages, the `rand` function generates a random number between 0 and 1]

There will be plenty more time to discuss conditional statements in later chapters. The above example should be pretty clear.

Ruby also includes a negated form of `if` called `unless` which goes something like

```
unless a > 5
  puts "a is less than or equal to 5"
else
  puts "a is greater than 5"
end
```

Generally speaking, Ruby keeps an `if` statement straight as long as the conditional (`if ...`) and the associated code block are on separate lines. If you have to smash everything together on one line, you'll need to place the `then` keyword after the conditional

```
if a < 5 then puts "#{a} less than 5" end
if a < 5 then puts "#{a} less than 5" else puts "#{a} greater than 5" end
```

Note that the `if` statement is also an expression; its value is the last line of the block executed. Therefore, the line above could also have been written as

```
puts(if a < 5 then "#{a} less than 5" else "#{a} greater than 5" end)
```

Ruby has also adopted the syntax from Perl where `if` and `unless` statements can be used as conditional modifiers *after* a statement. For example

```
puts "#{a} less than 5" if a < 5
puts "Cheese sandwich" unless a == 4
```

`while` behaves as it does in other languages -- the code block that follows is run zero or more times, as long as the conditional is true

```
while a > 5
  a = 10*rand
end
```

And like `if`, there is also a negated version of `while` called `until` which runs the code block *until* the condition is true.

Finally there is the `case` statement which we'll just include here with a brief example. `case` is actually a very powerful super version of the `if ... elsif...` system

```
a = (10*rand).round
#a = rand(11) would do the same

case a
when 0..5
  puts "#{a}: Low"
when 6
  puts "#{a}: Six"
else
  puts "#{a}: Cheese toast!"
end
```

There are some other interesting things going on in this example, but here the `case` statement is the center of attention.

Writing functions

In keeping with Ruby's all-object-oriented-all-the-time design, functions are typically referred to as methods. No difference. We'll cover methods in much more detail when we get to objects and classes. For now, basic method writing looks something like this:

```
# Demonstrate a method with func1.rb
```

```
def my_function( a )
  puts "Hello, #{a}"
  return a.length
end

len = my_function( "Giraffe" )
puts "My secret word is #{len} long"
```

```
$ func1.rb
Hello, Giraffe
My secret word is 7 long
```

Methods are defined with the `def` keyword, followed by the function name. As with variables, local and class methods should start with a lower case letter.

In this example, the function takes one argument (`a`) and returns a value. Note that the input arguments aren't typed (i.e. `a` need not be a string) ... this allows for great flexibility but can also cause a lot of trouble. The function also returns a single value with the `return` keyword. Technically this isn't necessary -- the value of the last line executed in the function is used as the return value -- but more often than not using `return` explicitly makes things clearer.

As with other languages, `Ruby` supports both default values for arguments and variable-length argument lists, both of which will be covered in due time. There's also support for code blocks, as discussed below.

Blocks

One very important concept in Ruby is the code block. It's actually not a particularly revolutionary concept -- any time you've written `if ... { ... }` in C or Perl you've defined a code block, but in Ruby a code block has some hidden secret powers...

Code blocks in Ruby are defined either with the keywords `do..end` or the curly brackets `{..}`

```
do
  print "I like "
  print "code blocks!"
end

{
  print "Me too!"
}
```

One very powerful usage of code blocks is that methods can take one as a parameter and *execute* it along the way.

[ed note: the Pragmatic Programmers actually want to point out that it's not very useful to describe it this way. Instead, the block of code behaves like a 'partner' to which the function occasionally hands over control]

The concept can be hard to get the first time it's explained to you. Here's an example:

```
$ irb --simple-prompt
>> 3.times { puts "Hi!" }
Hi!
Hi!
Hi!
=> 3
```

Surprise! You always thought 3 was just a number, but it's actually an object (of type `Fixnum`) As its an object, it has a member function `times` which takes a block as a parameter. The function runs the block 3 times.

Blocks can actually receive parameters, using a special notation `|..|`. In this case, a quick check of the documentation for `times` shows it will pass a single parameter into the block, indicating which loop it's on:

```
$ irb --simple-prompt
>> 4.times { |x| puts "Loop number #{x}" }
Loop number 0
Loop number 1
```

```
Loop number 2
Loop number 3
=> 4
```

The `times` function passes a number into the block. The block gets that number in the variable `x` (as set by the `|x|`), then prints out the result.

Functions interact with blocks through the `yield`. Every time the function invokes `yield` control passes to the block. It only comes back to the function when the block finishes. Here's a simple example:

```
# Script block2.rb

def simpleFunction
  yield
  yield
end

simpleFunction { puts "Hello!" }
```

```
$ block2.rb
Hello!
Hello!
```

The `simpleFunction` simply yields to the block twice -- so the block is run twice and we get two times the output. Here's an example where the function passes a parameter to the block:

```
# Script block1.rb

def animals
  yield "Tiger"
  yield "Giraffe"
end

animals { |x| puts "Hello, #{x}" }
```

```
$ block1.rb
Hello, Tiger
Hello, Giraffe
```

It might take a couple of reads through to figure out what's going on here. We've defined the function "animals" -- it expects a code block. When executed, the function calls the code block twice, first with the parameter "Tiger" then again with the parameter "Giraffe". In this example, we've written a simple code block which just prints out a greeting to the animals. We could write a different block, for example:

```
animals { |x| puts "It's #{x.length} characters long!" }
```

which would give:

```
It's 5 characters long!
It's 7 characters long!
```

Two completely different results from running the same function with two different blocks.

There are many powerful uses of blocks. One of the first you'll come across is the `each` function for arrays -- it runs a code block once for each element in the array -- it's great for iterating over lists.

Ruby is really, really object-oriented

Ruby is very object oriented. Everything is an object -- even things you might consider constants. This also means that the vast majority of what you might consider "standard functions" aren't floating around in some library somewhere, but are instead methods of a given variable.

Here's one example we've already seen:

```
3.times { puts "Hi!" }
```

Even though 3 might seem like just a constant number, it's in fact an instance of the class `Fixnum` (which inherits from the class `Numeric` which inherits from the class `Object`). The method `times` comes from `Fixnum` and does just what it claims to do.

Here are some other examples

```
$ irb --simple-prompt
>> 3.abs
=> 3
>> -3.abs
=> 3
>> "giraffe".length
=> 7
>> a = "giraffe"
=> "giraffe"
>> a.reverse
=> "effarig"
```

There will be lots of time to consider how object-oriented design filters through Ruby in the coming chapters.

Retrieved from "http://en.wikibooks.org/w/index.php?title=Ruby_Programming/Ruby_basics&oldid=2209947"

Ruby Programming/Data types

Ruby Data Types

As mentioned in the previous chapter, everything in Ruby is an object. Everything has a class. Don't believe me? Try running this bit of code:

```
h = {"hash?" => "yep, it\'s a hash!", "the answer to everything" => 42,
puts "Stringy string McString!".class
puts 1.class
puts nil.class
puts h.class
puts :symbol.class
```

Output:

```
String
Fixnum
NilClass
Hash
Symbol
```

See? Everything is an object. Every object has a method called `class` that returns that object's class. You can call methods on pretty much anything. Earlier you saw an example of this in the form of `3.times`. (Technically when you call a method you're sending a message to the object, but I'll leave the significance of that for later.)
Something that makes this extreme object oriented-ness very fun for me is the fact that all classes are open, meaning you can add variables and methods to a class at any time during the execution of your code. This, however, is a discussion of datatypes.

Constants

We'll start off with constants because they're simple. Two things to remember about constants:
1. Constants start with capital letters. `Constant` is a constant. `constant` is not a constant.
2. You can change the values of constants, but Ruby will give you a warning. (Silly, I know... but what can you do?)
Congrats. Now you're an expert on Ruby constants.

Symbols

So did you notice something weird about that first code listing? "What the heck was that colon thingy about?" Well, it just so happens that Ruby's object oriented ways have a cost: lots of objects make for slow code. Every time you type a string, Ruby makes a new object. Regardless of whether two strings are identical, Ruby treats every instance as a new object. You could have "live long and prosper" in your code once and then again later on and Ruby wouldn't even realize that they're pretty much the same thing. Here is

a sample irb session which demonstrates this fact :

irb> "live long and prosper".object_id
=> -507772268
irb> "live long and prosper".object_id
=> -507776538

Notice that the object ID returned by irb Ruby is different even for the same two strings.

To get around this memory hoggishness, Ruby has provided "symbols." Symbols are lightweight objects best used for comparisons and internal logic. If the user doesn't ever see it, why not use a symbol rather than a string? Your code will thank you for it. Let us try running the above code using symbols instead of strings :

irb> :my_symbol.object_id
=> 150808
irb> :my_symbol.object_id
=> 150808

Symbols are denoted by the colon sitting out in front of them, like so: `:symbol_name`

Hashes

Hashes are like dictionaries, in a sense. You have a key, a reference, and you look it up to find the associated object, the definition.

The best way to illustrate this, I think, is simply to demonstrate:

```
hash = {:leia => "Princess from Alderaan", :han => "Rebel without a cause", :luke => "Farmboy turned Jed
puts hash[:leia]
puts hash[:han]
puts hash[:luke]
```

This code will print out "Princess from Alderaan", "Rebel without a cause", and "Farmboy turned Jedi" in consecutive order. I could have also written this like so:

```
hash = {:leia => "Princess from Alderaan", :han => "Rebel without a cause", :luke => "Farmboy turned Jed
hash.each do |key, value|
     puts value
end
```

This code cycles through each element in the hash, putting the key in the `key` variable and the value in the `value` variable, which is then printed out.

If I wanted to be verbose about defining my hash, I could have even written it like this:

```
hash = Hash.[](:leia => "Princess from Alderaan", :han => "Rebel without a cause", :luke => "Farmboy tur
hash.each do |key, value|
     puts value
end
```

If I felt like offing Luke, I could do something like this:

```
hash.delete(:luke)
```

Now Luke's no longer in the hash. Or lets say I just had a vendetta against farmboys in general. I could do this:

```
hash.delete_if {|key, value| value.downcase.match("farmboy")}
```

This looks at each key-value pair and deletes it if the block of code following it returns `true`. In the block of code you see there, I made the value lowercase (in case the farmboys decided to start doing stuff like "FaRmBoY!1!") and then checked to see if "farmboy" matched anything in its contents. I could have used a regular expression, but that's another story.

If I felt like adding Lando into the mix, I'd just assign a new value to the hash like so:

```
hash[:lando] = "Dashing and debonair city administrator."
```

If I felt like measuring this hash, I'd just do `hash.length`. If I felt like looking at keys, I'd just use the `inspect` method to return the hash's keys as an array. Speaking of which...

Arrays

Arrays are a lot like hashes, but the keys are always consecutive numbers. Also, the first key in an array is always 0. So in an array with 5 items, the last element would be found at `array[4]` and the first element would be found at `array[0]`. In addition, all the methods you just learned with hashes can also be applied to arrays. Here are two ways to create arrays:

```
array1 = ["hello", "this", "is", "an", "array!"]
array2 = []
array2 << "This"
array2 << "is"
array2 << "also"
array2 << "an"
array2 << "array!"
```

As you may have guessed, the `<<` operator pushes values onto the end of an array. So if I were to write `puts array2[4]` after declaring those two arrays, the computer would print out `array!` Of course, if I felt like simultaneously getting `array!` and deleting it from the array, I could just `pop` it off. The `pop` method returns the last element in an array and then deletes it from the array. So if I ran this:

```
string = array2.pop
```

Then `string` would hold `array!` and `array2` would be one element shorter.
If I kept doing this, array2 wouldn't hold any elements. I can check for this condition by calling the `empty?` method. For example, the following bit of code moves all the elements from one array to another:

```
array1 << array2.pop until array2.empty?
```

Here's something that really excites me: arrays can be subtracted and added to each other. I don't know about any other languages, but I know that Java would make a fuss if I tried the following bit of code:

```
array3 = array - array2
array4 = array + array2
```

Now `array3` holds all the elements that `array` did except for the ones that also happened to be in `array2`. So all the elements of `array` minus the elements of `array2` are now held in `array3`. `array4` now holds all the elements of both `array` and `array2`.
You wanna' see if `array` has something in it? Just ask it with the `include?` method, like so:
`array.include?("Is this in here?")`
If you just wanted to turn the whole array into a string, you'd do something like this:

```
string = array2.join(" ")
```

Using the array we just declared, `string` now holds `This is also an array!` If we had wanted something more along the lines of `Thisisalsoanarray!` we could have called the `join` method without any arguments, like so:

```
string = array2.join
```

Strings

I would recommend reading the chapters on strings and alternate quotes now if you haven't already. This chapter is going to cover some pretty spiffy things with strings and just assume that you already know the information in these two chapters.
In Ruby, there are some pretty cool built-in functions where strings are concerned. For example, you can multiply them. `"Danger, Will Robinson!" * 5` yields `Danger, Will Robinson!Danger, Will Robinson!Danger, Will Robinson!Danger, Will Robinson!` You can also compare strings like so: `"a" < "b"`
This comparison will yield `true`. When you compare two strings like this, you are really comparing the ASCII values of the characters. To find out what the ASCII value of a character is, use the following code:

```
puts ?A
```

Or, if you're using Ruby version 1.9 (http://stackoverflow.com/questions/1270209/getting-an-ascii-character-code-in-ruby-fails) or better, use the **ord** method:

```
puts "A".ord
```

This will return 65, as this is the ASCII value of "A". Just replace `A` with whatever character you want to inquire about. To do the opposite conversion, use the `chr` method. So `65.chr` would return `A`.
Aside from these weird little operations, we also have the venerable "concatenate" operation. It's the same as in most other languages: `+`
So `"Hi, this is " + "a concatenated string!"` would return `Hi, this is a concatenated string!`
You can also do some interpolation, for handling pesky string variables without using the concatenate operator. `string1` and `string2` in the following chunk of code are identical.

```
thing1 = "Red fish, "
thing2 = "blue fish."
string1 = thing1 + thing2 + " And so on and so forth."
string2 = "#{thing1 + thing2} And so on and so forth."
```

If you wanted to step through each one of the letters in `thing1`, you would use the `scan` method like so:

```
thing1.scan(/./) {|letter| puts letter}
```

This would print out each letter in `thing1`. But what's with that weird "`/./`" thing in the parameter? That, my friend, is called a "regular expression." They're helpful little buggers, quite powerful, but they're also outside the scope of this discussion. All you need to know for now is that `/./` is "reg-ex" speak for "every individual character." If I had typed `/../` then it would have gone through every chunk of two characters. Another use for regular expressions can be found with the `=~` operator. You can check to see if a string matches a regular expression using this. For example:

```
puts "Yeah, there's a number in this one." if "C3-P0, human-cyborg relations" =~ /[0-9]/
```

This will print out `Yeah, there's a number in this one.` The `match` method works much the same way, except it can accept a string as a parameter as well. This is helpful if you're getting regular expressions from a source outside the code. Here's what it looks like in action:

```
puts "Yep, they mentioned Jabba in this one." if "Jabba the Hutt".match("Jabba")
```

Alright, that's enough about the regular expressions. Even though you can use regular expressions with these next two methods, we'll just use regular old strings. Lets pretend you work at the Ministry of Truth and you need to replace a word in a string with another word. You'd do it like this:

```
string1 = "2 + 2 = 4"
string2 = string1.sub("4", "5")
```

Now `string2` contains `2 + 2 = 5`. But what if the string contains lots of lies like the one you just corrected? `sub` only replaces the first occurrence of a word! I guess you could cycle through the string using `match` and a `while` loop, but there's a much more efficient way to do things. Observe:

```
winston = %q{   Down with Big Brother!
                Down with Big Brother!
                Down with Big Brother!
                Down with Big Brother!
                Down with Big Brother!}
winston.gsub("Down with", "Long live")
```

Big Brother would be proud. `gsub` is the "global substitute" function. Every occurrence of "Down with" has now been replaced with "Long live" so now `winston` is only proclaiming its love for Big Brother, not its disdain thereof.
On that happy note, lets move on to integers and floats. (If you want to know more methods you can use with strings, look at the end of this chapter for a quick reference table.)

Numbers (Integers and Floats)

You can skip this paragraph if you know all the standard number operators. For those who don't, here's a crash course. `+` adds two numbers together. `-` subtracts them. `/` divides. `*` multiplies. `%` returns the remainder of two divided numbers.

Alright, integers are numbers with no decimal place. Floats are numbers with decimal places. `10 / 3` yields `3` because dividing two integers yields an integer. Since integers have no decimal places all you get is `3`. If you tried `10.0 / 3` you would get `3.33333...` If you have even one float in the mix you get a float back. Capice?

Alright, let's get down to the fun part. Everything in Ruby is an object, let me reiterate. That means that pretty much everything has at least one method. Integers and floats are no exception. First I'll show you some integer methods.
Here we have the venerable `times` method. Use it whenever you want to do something more than once. Examples:

```
puts "I will now count to 99..."
100.times {|number| puts number}
5.times {puts "Guess what?"}
puts "I'm done!"
```

This will print out the numbers 0 through 99, print out `Guess what?` five times, then say `I'm done!` It's basically a simplified `for` loop. It's a little slower than a `for` loop by a few hundredths of a second or so; keep that in mind if you're ever writing Ruby code for NASA. ;-)

Alright, we're nearly done, six more methods to go. Here are three of them:

```
# First a visit from The Count...
1.upto(10) {|number| puts "#{number} Ruby loops, ah-ah-ah!"}

# Then a quick stop at NASA...
puts "T-minus..."
10.downto(1) {|x| puts x}
puts "Blast-off!"

# Finally we'll settle down with an obscure Schoolhouse Rock video...
5.step(50, 5) {|x| puts x}
```

Alright, that should make sense. In case it didn't, `upto` counts up from the number it's called from to the number passed in its parameter. `downto` does the same, except it counts down instead of up. Finally, `step` counts from the number its called from to the first number in its parameters by the second number in its parameters. So `5.step(25, 5) {|x| puts x}` would output every multiple of five starting with five and ending at twenty-five.

Time for the last three:

```
string1 = 451.to_s
string2 = 98.6.to_s
int = 4.5.to_i
float = 5.to_f
```

`to_s` converts floats and integers to strings. `to_i` converts floats to integers. `to_f` converts integers to floats. There you have it. All the data types of Ruby in a nutshell. Now here's that quick reference table for string methods I promised you.

Additional String Methods

```
# Outputs 1585761545
"Mary J".hash

# Outputs "concatenate"
"concat" + "enate"

# Outputs "Washington"
"washington".capitalize

# Outputs "uppercase"
"UPPERCASE".downcase
```

```
# Outputs "LOWERCASE"
"lowercase".upcase

# Outputs "Henry VII"
"Henry VIII".chop

# Outputs "rorriM"
"Mirror".reverse

# Outputs 810
"All Fears".sum

# Outputs cRaZyWaTeRs
"CrAzYwAtErS".swapcase

# Outputs "Nexu" (next advances the word up one value, as if it were a number.)
"Next".next

# After this, nxt == "Neyn" (to help you understand the trippiness of next)
nxt = "Next"
20.times {nxt = nxt.next}
```

Ruby Programming/Writing methods

Defining Methods

Methods are defined using the def keyword and ended with the end keyword. Some programmers find the Methods defined in Ruby very similar to those in Python.

```
def myMethod
end
```

To define a method that takes in a value, you can put the local variable name in parentheses after the method definition. The variable used can only be accessed from inside the method scope.

```
def myMethod(msg)
    puts msg
end
```

If multiple variables need to be used in the method, they can be separated with a comma.

```
def myMethod(msg, person)
    puts "Hi, my name is " + person + ". Some information about mys
end
```

Any object can be passed through using methods.

```
def myMethod(myObject)
    if(myObject.is_a?(Integer))
          puts "Your Object is an Integer"
    end
    #Check to see if it defined as an Object that we created
    #You will learn how to define Objects in a later section
    if(myObject.is_a?(MyObject))
          puts "Your Object is a MyObject"
    end
end
```

The return keyword can be used to specify that you will be returning a value from the method defined.

```
def myMethod
    return "Hello"
end
```

It is also worth noting that ruby will return the last expression evaluated, so this is functionally equivalent to the previous method.

```
def myMethod
    "Hello"
end
```

Some of the Basic Operators can be overridden using the def keyword and the operator that you wish to override.

```
def ==(oVal)
    if oVal.is_a?(Integer)
        #@value is  a variable defined in the class where this meth
        #This will be covered in a later section when dealing with
        if(oVal == @value)
            return true
        else
            return false
        end
    end
end
```

Retrieved from "http://en.wikibooks.org/w/index.php?title=Ruby_Programming/Writing_methods&oldid=2166105"

Ruby Programming/Classes and objects

Ruby Classes

As stated before, everything in Ruby is an object. Every object has a class. To find the class of an object, simply call that object's `class` method. For example, try this:

```ruby
puts "This is a string".class
puts 9.class
puts ["this","is","an","array"].class
puts {:this => "is", :a => "hash"}.class
puts :symbol.class
```

Anyhow, you should already know this. What you don't know however, is how to make your own classes and extend Ruby's classes.

Creating Instances of a Class

An instance of a class is an object that has that class. For example, `"chocolate"` is an instance of the String class. You already know that you can create strings, arrays, hashes, numbers, and other built-in types by simply using quotes, brackets, curly braces, etc., but you can also create them via the `new` method. For example, `my_string = ""` is the same as `my_string = String.new`. Every class has a `new` method: arrays, hashes, integers, whatever. When you create your own classes, you'll use the `new` method to create instances.

Creating Classes

Classes represent a type of an object, such as a book, a whale, a grape, or chocolate. Everybody likes chocolate, so let's make a chocolate class:

```ruby
class Chocolate
  def eat
    puts "That tasted great!"
  end
end
```

Let's take a look at this. Classes are created via the `class` keyword. After that comes the name of the class. All class names must start with a Capital Letter. By convention, we use CamelCase for class name. So we would create classes like PieceOfChocolate, but not like Piece_of_Chocolate.

The next section defines a class method. A class method is a method that is defined for a particular class. For example, the String class has the `length` method:

```ruby
# outputs "5"
```

```
puts "hello".length
```

To call the `eat` method of an instance of the Chocolate class, we would use this code:

```
my_chocolate = Chocolate.new
my_chocolate.eat # outputs "That tasted great!"
```

You can also call a method by using `send`

```
"hello".send(:length) # outputs "5"
my_chocolate.send(:eat) # outputs "That tasted great!"
```

However, using `send` is rare unless you need to create a dynamic behavior, as we do not need to specify the name of the method as a literal - it can be a variable.

Self

Inside a method of a class, the pseudo-variable `self` (a pseudo-variable is one that cannot be changed) refers to the current instance. For example:

```
class Integer
  def more
    return self + 1
  end
end
3.more # -> 4
7.more # -> 8
```

Class Methods

You can also create methods that are called on a class rather than an instance. For example:

```
class Strawberry
  def Strawberry.color
    return "red"
  end

  def self.size
    return "kinda small"
  end

  class << self
    def shape
      return "strawberry-ish"
    end
  end
```

```
end
Strawberry.color # -> "red"
Strawberry.size  # -> "kinda small"
Strawberry.shape # -> "strawberry-ish"
```

Note the three different constructions: `ClassName.method_name` and `self.method_name` are essentially the same - outside of a method definition in a class block, `self` refers to the class itself. The latter is preferred, as it makes changing the name of the class much easier. The last construction, `class << self`, puts us in the context of the class's "meta-class" (sometimes called the "eigenclass"). The meta-class is a special class that the class itself belongs to. However, at this point, you don't need to worry about it. All this construct does is allow us to define methods without the `self.` prefix.

Retrieved from "http://en.wikibooks.org/w/index.php?title=Ruby_Programming/Classes_and_objects&oldid=2262479"

Ruby Programming/Syntax/Lexicology

Identifiers

An **identifier** is a name used to identify a variable, method, or class.

As with most languages, valid identifiers consist of alphanumeric characters (`A-Za-z0-9`) and underscores (_), but may not begin with a digit (`0-9`). Additionally, identifiers that are *method* names may end with a question mark (`?`), exclamation point (`!`), or equals sign (`=`).

There are no arbitrary restrictions to the length of an identifier (i.e. it may be as long as you like, limited only by your computer's memory). Finally, there are reserved words which may not be used as identifiers.

Examples:

```
foobar
ruby_is_simple
```

Comments

Line comments run from a bare '#' character to the end of the line. There are no multi-line comments.

Examples:

```
# this line does nothing
print "Hello" # this line prints "Hello"
```

Embedded Documentation

Example:

```
=begin
Everything between a line beginning with `=begin' down to
one beginning with `=end' will be skipped by the interpreter.
These reserved words must begin in column 1.
=end
```

Reserved Words

The following words are reserved in Ruby:

```
__FILE__  and    def       end     in      or      self   unless
__LINE__  begin  defined?  ensure  module  redo    super  until
BEGIN     break  do        false   next    rescue  then   when
END       case   else      for     nil     retry   true   while
alias     class  elsif     if      not     return  undef  yield
```

You can find some examples of using them here (http://ruby-doc.org/docs/keywords/1.9) .

Expressions

Example: true

```
(1 + 2) * 3
foo()
if test then okay else not_good end
```

All variables, literals, control structures, etcetera are expressions. Using these together is called a program. You can divide expressions with newlines or semicolons (;) — however, a newline with a preceding backslash (\) is continued to the following line.

Since in Ruby control structures are expressions as well, one can do the following:

```
foo = case 1
      when 1
        true
      else
        false
      end
```

The above equivalent in a language such as C would generate a syntax error since control structures are not expressions in the C language.

Ruby Programming/Syntax/Variables and Constants

A variable in Ruby can be distinguished by the characters at the start of its name. There's no restriction to the length of a variable's name (with the exception of the heap size).

Local Variables

Example:

```
foobar
```

A variable whose name begins with a lowercase letter (a-z) or underscore (_) is a local variable or method invocation.

A local variable is only accessible from within the block of its initialization. For example:

```
i0 = 1
loop {
  i1 = 2
  puts defined?(i0)     # true; "i0" was initialized in the ascendant block
  puts defined?(i1)     # true; "i1" was initialized in this block
  break
}
puts defined?(i0)       # true; "i0 was initialized in this block
puts defined?(i1)       # false; "i1" was initialized in the loop
```

Instance Variables

Example:

```
@foobar
```

A variable whose name begins with '`@`' is an instance variable of `self`. An instance variable belongs to the object itself. Uninitialized instance variables have a value of `nil`.

Class Variables

A class variable is shared by all instances of a class. Example:

```
@@foobar
```

An important note is that the class variable is shared by all the descendants of the class. Example:

```
class Parent
  @@foo = "Parent"
 end
```

```
class Thing1 < Parent
  @@foo = "Thing1"
end
class Thing2 < Parent
  @@foo = "Thing2"
end
>>Parent.class_eval("@@foo")
=>"Thing2"
>>Thing1.class_eval("@@foo")
=>"Thing2"
>>Thing2.class_eval("@@foo")
=>"Thing2"
>>Thing2.class_variables
=>[]
Parent.class_variables
=>[:@@foo]
```

This shows us that all our classes were changing the same variable. Class variables behave like global variables which are visible only in the inheritance tree. Because Ruby resolves variables by looking up the inheritance tree *first*, this can cause problems if two subclasses both add a class variable with the same name.

Global Variables

Example:

```
$foobar
```

A variable whose name begins with '$' has a global scope; meaning it can be accessed from anywhere within the program during runtime.

Constants

Usage:

```
FOOBAR
```

A variable whose name begins with an uppercase letter (A-Z) is a constant. A constant can be reassigned a value after its initialization, but doing so will generate a warning. Every class is a constant.

Trying to access an uninitialized constant raises the NameError exception.

How constants are looked up

Constants are looked up based on your scope. For example

```
class A
  A2 = 'a2'
  class B
    def go
      A2
    end
  end
end
instance_of_b = A::B.new
a2 = A::A2
```

Pseudo Variables

`self`

Execution context of the current method.

`nil`

The sole-instance of the `NilClass` class. Expresses nothing.

`true`

The sole-instance of the `TrueClass` class. Expresses true.

`false`

The sole-instance of the `FalseClass` class. Expresses false.

`$1, $2 ... $9`

These are contents of capturing groups for regular expression matches. They are local to the **current thread and stack frame**!

(**nil** also is considered to be **false**, and every other value is considered to be **true** in Ruby.) The value of a pseudo variable cannot be changed. Substitution to a pseudo variable causes an exception to be raised.

Pre-defined Variables

Name	Aliases	Description
`$!`	`$ERROR_INFO`[1]	The exception information message set by the last 'raise' (last exception thrown).
`$@`	`$ERROR_POSITION`[1]	Array of the backtrace of the last exception thrown.
`$&`	`$MATCH`[1]	The string matched by the last successful pattern match in this scope.
`$``	`$PREMATCH`[1]	The string to the left of the last successful match.
`$'`	`$POSTMATCH`[1]	The string to the right of the last successful match.
`$+`	`$LAST_PAREN_MATCH`[1]	The last bracket matched by the last successful match.
`$1` to `$9`		The Nth group of the last successful regexp match.
`$~`	`$LAST_MATCH_INFO`[1]	The information about the last match in the current scope.
`$=`	`$IGNORECASE`[1]	The flag for case insensitive, nil by default (deprecated).
`$/`	`$INPUT_RECORD_SEPARATOR`[1], `$RS`[1] or `$-0`	The input record separator, newline by default.
`$\`	`$OUTPUT_RECORD_SEPARATOR`[1] or `$ORS`[1]	The output record separator for the print and IO#write. Default is nil.
`$,`	`$OUTPUT_FIELD_SEPARATOR`[1] or `$OFS`[1]	The output field separator for the print and Array#join.

$;	$FIELD_SEPARATOR[1], $FS[1] or $-F	The default separator for String#split.
$.	$INPUT_LINE_NUMBER[1] or $NR[1]	The current input line number of the last file that was read.
$<	$DEFAULT_INPUT[1]	An object that provides access to the concatenation of the contents of all the files given as command-line arguments, or $stdin (in the case where there are no arguments). Read only.
$FILENAME		Current input file from $<. Same as $<.filename.
$>	$DEFAULT_OUTPUT[1]	The destination of output for Kernel.print and Kernel.printf. The default value is $stdout.
$_	$LAST_READ_LINE[1]	The last input line of string by gets or readline.
$0		Contains the name of the script being executed. May be assignable.
$*	$ARGV[1]	Command line arguments given for the script. Also known as ARGV
$$	$PROCESS_ID[1], $PID[1] or Process.pid	The process number of the Ruby running this script.
$?	$CHILD_STATUS[1]	The status of the last executed child process.
$:	$LOAD_PATH	Load path for scripts and binary modules by load or require.
$"	$LOADED_FEATURES or $-I	The array contains the module names loaded by require.
$stderr		The current standard error output.
$stdin		The current standard input.
$stdout		The current standard output.
$-d	$DEBUG	The status of the -d switch. Assignable.
$-K	$KCODE	Character encoding of the source code.
$-v	$VERBOSE	The verbose flag, which is set by the -v switch.
$-a		True if option -a ("autosplit" mode) is set. Read-only variable.
$-i		If in-place-edit mode is set, this variable holds the extension, otherwise nil.
$-l		True if option -l is set ("line-ending processing" is on). Read-only variable.
$-p		True if option -p is set ("loop" mode is on). Read-only variable.
$-w		True if option -w is set.

The use of cryptic two-character $? expressions is a thing that people will frequently complain about, dismissing Ruby as just another perl-ish line-noise language. Keep this chart handy. Note, a lot of these are useful when working with regexp code. Part of the standard library is "English" which defines longer names to replace the two-character variable names to make code more readable. The defined names are also listed in the table. To include these names, just require the English library as follows.[1]

Without 'English':

```
$\ = ' -- '
"waterbuffalo" =~ /buff/
print $", $', $$, "\n"
```

With English:

```
require "English"

$OUTPUT_FIELD_SEPARATOR = ' -- '
"waterbuffalo" =~ /buff/
print $LOADED_FEATURES, $POSTMATCH, $PID, "\n"
```

Pre-defined Constants

Note that there are some pre-defined constants at parse time, as well, namely

```
__FILE__ (current file)
```

and

```
__LINE__ (current line)
```

Notes

1. ↑ ***a b c d e f g h i j k l m n o p q r s t u v w x y z*** English.rb (http://ruby-doc.org/stdlib/libdoc/English/rdoc/index.html) from the Ruby 1.9.2 Standard Library Documentation (http://ruby-doc.org/stdlib/)

Ruby Programming/Syntax/Literals

Numerics

```
123                     # Fixnum
-123                    # Fixnum (signed)
1_123                   # Fixnum (underscore is ignored)
-543                    # Negative Fixnum
123_456_789_123_456_789 # Bignum
123.45                  # Float
1.2e-3                  # Float
0xaabb                  # (Hexadecimal) Fixnum
0377                    # (Octal) Fixnum
-0b1010                 # (Binary [negated]) Fixnum
0b001_001               # (Binary) Fixnum
?a                      # ASCII character code for 'a' (97)
?\C-a                   # Control-a (1)
?\M-a                   # Meta-a (225)
?\M-\C-a                # Meta-Control-a (129)
```

Strings

Examples:

```
"this is a string"
=> "this is a string"

"three plus three is #{3+3}"
=> "three plus three is 6"

foobar = "blah"
"the value of foobar is #{foobar}"
=> "the value of foobar is blah"

'the value of foobar is #{foobar}'
=> "the value of foobar is \#{foobar}"
```

A string expression begins and ends with a double or single-quote mark. Double-quoted string expressions are subject to backslash notation and interpolation. A single-quoted string expression isn't; except for \' and \\.

Backslash Notation

Also called *escape characters* or *escape sequences*, they are used to insert special characters in a string.

Example:

```
"this is a\ntwo line string"
"this string has \"quotes\" in it"
```

Escape Sequence	Meaning
\n	newline (0x0a)
\s	space (0x20)
\r	carriage return (0x0d)
\t	tab (0x09)

\v	vertical tab (0x0b)
\f	formfeed (0x0c)
\b	backspace (0x08)
\a	bell/alert (0x07)
\e	escape (0x1b)
\nnn	character with octal value nnn
\xnn	character with hexadecimal value nn
\unnnn	Unicode code point U+nnnn (Ruby 1.9 and later)
\cx	control-x
\C-x	control-x
\M-x	meta-x
\M-\C-x	meta-control-x
\x	character x itself (\" a single quote, for example)

For characters with decimal values, you can do this:

```
"" << 197 # add decimal value 304 to a string
```

or embed them thus:

```
"#{197.chr}"
```

Interpolation

Interpolation allows Ruby code to appear within a string. The result of evaluating that code is inserted into the string:

```
"1 + 2 = #{1 + 2}"    #=> "1 + 2 = 3"
```

```
#{expression}
```

The expression can be just about any Ruby code. Ruby is pretty smart about handling string delimiters that appear in the code and it generally does what you want it to do. The code will have the same side effects as it would outside the string, including any errors:

```
"the meaning of life is #{1/0}"
=> divided by 0 (ZeroDivisionError)
```

The % Notation

There is also a Perl-inspired way to quote strings: by using % (percent character) and specifying a delimiting character, for example:

```
%{78% of statistics are "made up" on the spot}
=> "78% of statistics are \"made up\" on the spot"
```

Any single non-alpha-numeric character can be used as the delimiter, `%[including these], %?or`

`these?, %~or even these things~`. By using this notation, the usual string delimiters " and ' can appear in the string unescaped, but of course the new delimiter you've chosen does need to be escaped. However, if you use `%(parentheses), %[square brackets], %{curly brackets}` or `%<pointy brackets>` as delimiters then those same delimiters can appear *unescaped* in the string as long as they are in *balanced* pairs:

```
%(string (syntax) is pretty flexible)
=> "string (syntax) is pretty flexible"
```

A modifier character can appear after the %, as in %q[], %Q[], %x[] - these determine how the string is interpolated and what type of object is produced:

Modifier	**Meaning**
%q[ ]	Non-interpolated String (except for \\ \[and \])
%Q[ ]	Interpolated String (default)
%r[ ]	Interpolated Regexp (flags can appear after the closing delimiter)
%s[ ]	Non-interpolated Symbol
%w[ ]	Non-interpolated Array of words, separated by whitespace
%W[ ]	Interpolated Array of words, separated by whitespace
%x[ ]	Interpolated shell command

Here are some more examples:

```
%Q{one\ntwo\n#{ 1 + 2 }}
=> "one\ntwo\n3"

%q{one\ntwo\n#{ 1 + 2 }}
=> "one\\ntwo\\n#{ 1 + 2 }"

%r{nemo}i
=> /nemo/i

%w{one two three}
=> ["one", "two", "three"]

%x{ruby --copyright}
=> "ruby - Copyright (C) 1993-2009 Yukihiro Matsumoto\n"
```

"Here document" notation

There is yet another way to make a string, known as a 'here document', where the delimiter itself can be any string:

```
string = <<END
on the one ton temple bell
a moon-moth, folded into sleep,
sits still.
END
```

The syntax begins with << and is followed immediately by the delimiter. To end the string, the delimiter appears alone on a line.

There is a slightly nicer way to write a here document which allows the ending delimiter to be indented by whitespace:

```
string = <<-FIN
           on the one ton temple bell
           a moon-moth, folded into sleep,
           sits still.
         FIN
```

To use non-alpha-numeric characters in the delimiter, it can be quoted:

```
string = <<-"."
           orchid breathing
           incense into
           butterfly wings.
         .
```

Here documents are interpolated, *unless you use single quotes around the delimiter*.

The rest of the line after the opening delimiter is not interpreted as part of the string, which means you can do this:

```
strings = [<<END, "short", "strings"]
a long string
END

=> ["a long string\n", "short", "strings"]
```

You can even "stack" multiple here documents:

```
string = [<<ONE, <<TWO, <<THREE]
  the first thing
ONE
  the second thing
TWO
  and the third thing
THREE

=> ["the first thing\n", "the second thing\n", "and the third thing\n"]
```

Arrays

An array is a collection of objects indexed by a non-negative integer. You can create an array object by writing `Array.new`, by writing an optional comma-separated list of values inside square brackets, or if the array will only contain string objects, a space-delimited string preceded by `%w`.

```
array_one   = Array.new
array_two   = []              # shorthand for Array.new
array_three = ["a", "b", "c"] # array_three contains "a", "b" and "c"
array_four  = %w[a b c]       # array_four also contains "a", "b" and "c"
```

```
array_three[0]                # => "a"
array_three[2]                # => "c"
array_four[0]                 # => "a"
#negative indices are counted back from the end
array_four[-2]                # => "b"
#[start, count] indexing returns an array of count objects beginning at index start
array_four[1,2]               # => ["b", "c"]
#using ranges. The end position is included with two periods but not with three
array_four[0..1]              # => ["a", "b"]
array_four[0...1]             # => ["a"]
```

The last method, using `%w`, is in essence shorthand for the `String` method `split` when the substrings are separated by whitespace only. In the following example, the first two ways of creating an array of strings are

functionally identical while the last two create very different (though both valid) arrays.

```
array_one   = %w'apple orange pear'           # => ["apple", "orange", "pear"]
array_two   = 'apple orange pear'.split       # => ["apple", "orange", "pear"]
array_one   == array_two                      # => true

array_three = %w'dog:cat:bird'                # => ["dog:cat:bird"]
array_four  = 'dog:cat:bird'.split(':')       # => ["dog", "cat", "bird"]
array_three == array_four                     # => false
```

Hashes

Hashes are basically the same as arrays, except that a hash not only contains values, but also keys pointing to those values. Each key can occur only once in a hash. A hash object is created by writing `Hash.new` or by writing an optional list of comma-separated `key => value` pairs inside curly braces.

```
hash_one   = Hash.new
hash_two   = {}                               # shorthand for Hash.new
hash_three = {"a" => 1, "b" => 2, "c" => 3} #=> {"a"=>1, "b"=>2, "c"=>3}
```

Usually Symbols are used for Hash keys (allows for quicker access), so you will see hashes declared like this:

```
hash_sym   = { :a => 1, :b => 2, :c => 3}   #=> {:b=>2, :c=>3, :a=>1}
hash_sym   = { a: 1, b: 2, c: 3}            #=> {:b=>2, :c=>3, :a=>1}
```

The latter form was introduced in Ruby 1.9.

Hash ordering

Note that with 1.8, iterating over hashes will iterate over the key value pairs in a "random" order. Beginning with 1.9, it will iterate over them in the order they were inserted. Note however, that if you reinsert a key (or change an existing key's value), that does not change that keys' order in the iteration.

```
>> a = {:a => 1, :b => 2, :c => 3}
=> {:a=>1, :b=>2, :c=>3}
>> a.keys # iterate over, show me the keys
=> [:a, :b, :c]
>> a[:b] = 4
> a.keys
=> [:a, :b, :c] # same order
>> a.delete(:b)
>> a[:b] = 4 # re insert now
=> 4
>> a.keys
=> [:a, :c, :b] # different order
```

Ranges

A **range** represents a subset of all possible values of a type, to be more precise, all possible values between a start value and an end value.

This may be:

- All integers between 0 and 5.
- All numbers (including non-integers) between 0 and 1, excluding 1.
- All characters between 't' and 'y'.

In Ruby, these ranges are expressed by:

```
0..5
0.0...1.0
't'..'y'
```

Therefore, ranges consist of a start value, an end value, and whether the end value is included or not (in this short syntax, using two . for including and three . for excluding).

A range represents a set of values, not a sequence. Therefore,

```
5..0
```

though syntactically correct, produces a range of length zero.

Ranges can only be formed from instances of the same class or subclasses of a common parent, which must be Comparable (implementing <=>).

Ranges are instances of the Range class, and have certain methods, for example, to determine whether a value is inside a range:

```
r = 0..5
puts r === 4  # => true
puts r === 7  # => false
```

For detailed information of all Range methods, consult the Range class reference.

Here (http://www.engineyard.com/blog/2010/iteration-shouldnt-spin-your-wheels/) is a tutorial on their use.

Retrieved from "http://en.wikibooks.org/w/index.php?title=Ruby_Programming/Syntax/Literals&oldid=2231569"

Ruby Programming/Syntax/Operators

Operators

1. Assignment

Assignment in Ruby is done using the equal operator "=". This is both for variables and objects, but since strings, floats, and integers are actually objects in Ruby, you're always assigning objects.

Examples:

```
myvar = 'myvar is now this string'
var = 321
dbconn = Mysql::new('localhost','root','password')
```

Self assignment

```
x = 1           #=>1
x += x          #=>2
x -= x          #=>0
x += 4          #=>x was 0 so x= + 4 # x is positive 4
x *= x          #=>16
x **= x         #=>18446744073709551616 # Raise to the power
x /= x          #=>1
```

A frequent question from C and C++ types is "How do you increment a variable? Where are ++ and -- operators?" In Ruby, one should use x+=1 and x-=1 to increment or decrement a variable.

```
x = 'a'
x.succ!         #=>"b" : succ! method is defined for String, but not for Integer types
```

Multiple assignments

Examples:

```
var1, var2, var3 = 10, 20, 30
puts var1           #=>var1 is now 10
puts var2           #=>var2 is now 20,var3...etc

myArray=%w(John Michel Fran Doug) # %w() can be used as syntactic sugar to simplify array creation
var1,var2,var3,var4=*myArray
puts var1           #=>John
puts var4           #=>Doug
```

```
names,school=myArray,'St. Whatever'
names               #=>["John", "Michel", "Fran", "Doug"]
school              #=>"St. Whatever"
```

Conditional assignment

```
x = find_something() #=>nil
x ||= "default"      #=>"default" : value of x will be replaced with "default", but only if x is nil or
x ||= "other"        #=>"default" : value of x is not replaced if it already is other than nil or false
```

Operator ||= is a shorthand form of the expression:

```
x = x || "default"
```

Operator ||= can be shorthand for code like:

```
x = "(some fallback value)" unless respond_to? :x or x
```

In same way &&= operator works:

```
x = get_node() #=>nil
x &&= x.next_node #=> nil : x will be set to x.next_node, but only if x is NOT nil or false
x = get_node() #=>Some Node
x &&= x.next_node #=>Next Node
```

Operator &&= is a shorthand form of the expression:

```
x = x && x.get_node()
```

Scope

In Ruby there's a local scope, a global scope, an instance scope, and a class scope.

Local Scope

Example:

```
var=2
4.times do |x|
  puts x=x*var
end
#=>0,2,4,6
puts x #=>undefined local variable or method `x' for main:Object (NameError)
```

This error appears because this x(toplevel) is not the x(local) inside the do..end block the x(local) is a local variable to the block, whereas when trying the puts x(toplevel) we're calling a x variable that is in the top level scope, and since there's not one, Ruby protests.

Global scope

```
4.times do |$global|
  $global=$global*var
end  #=>0,2,4,6 last assignment of $global is 6
puts $global
#=> 6
```

This works because prefixing a variable with a dollar sign makes the variable a global.

Instance scope

Within methods of a class, you can share variables by prefixing them with an @.

```
class A
  def setup
    @instvar = 1
```

```
  end
  def go
    @instvar = @instvar*2
    puts @instvar
  end
end
instance = A.new
instance.setup
instance.go
#=> 2
instance.go
#=> 4
```

Class scope

A class variable is one that is like a "static" variable in Java. It is shared by all instances of a class.

```
class A
  @@instvar = 1
  def go
    @@instvar = @@instvar*2
    puts @@instvar
  end
end
instance = A.new
instance.go
#=> 2
instance = A.new
instance.go
#=> 4 -- variable is shared across instances
```

Here's a demo showing the various types:

```
$variable
 class Test
   def initialize(arg1='kiwi')
     @instvar=arg1
     @@classvar=@instvar+' told you so!!'
     localvar=@instvar
   end
   def print_instvar
     puts @instvar
   end
   def print_localvar
     puts @@classvar
     puts localvar
   end
 end
 var=Test.new
 var.print_instvar              #=>"kiwi", it works because a @instance_var can be accesed inside the cl
 var.print_localvar             #=>undefined local variable or method `localvar' for #<Test:0x2b36208 @i
```

This will print the two lines "kiwi" and "kiwi told you so!!", then FAIL! with a undefined local variable or method `localvar' for #<Test:0x2b36208 @instvar="kiwi"> (NameError). Why? well, in the scope of the method print_localvar there doesn't exists localvar, it exists in method initialize(until GC kicks it out). On the other hand ,class variables '@@classvar' and '@instvar' are in scope across the entire class and, in the case of @@class variables, across the children classes.

```
class SubTest < Test
  def print_classvar
    puts @@classvar
  end
end
newvar=SubTest.new              #newvar is created and it has @@classvar with the same value as the var
newvar.print_classvar           #=>kiwi told you so!!
```

Class variables have the scope of parent class AND children, these variables can live across classes, and can

be affected by the children actions ;-)

```
class SubSubTest < Test
  def print_classvar
    puts @@classvar
  end
  def modify_classvar
    @@classvar='kiwi kiwi waaai!!'
  end
end
subtest=SubSubTest.new
subtest.modify_classvar          #lets add a method that modifies the contents of @@classvar in  SubSub
subtest.print_classvar
```

This new child of Test also has @@classvar with the original value newvar.print_classvar. The value of @@classvar has been changed to 'kiwi kiwi waaai!!' This shows that @@classvar is "shared" across parent and child classes.

Default scope

When you don't enclose your code in any scope specifier, ex:

```
@a = 33
```

it affects the default scope, which is an object called "main".

For example, if you had one script that says

```
@a = 33
require 'other_script.rb'
```

and other_script.rb says

```
puts @a
 #=> 33
```

They could share variables.

Note however, that the two scripts don't share local variables.

Local scope gotchas

Typically when you are within a class, you can do as you'd like for definitions, like.

```
class A
  a = 3
  if a == 3

    def go
       3
    end
  else
    def go
       4
    end
  end

end
```

And also, procs "bind" to their surrounding scope, like

```
a = 3
b = proc { a }
b.call # 3 -- it remembered what a was
```

However, the "class" and "def" keywords cause an *entirely new* scope.

```
class A
  a = 3
  def go
     return a # this won't work!
  end
end
```

You can get around this limitation by using define_method, which takes a block and thus keeps the outer scope (note that you can use any block you want, to, too, but here's an example).

```
class A
   a = 3
  define_method(:go) {
      a
  }
end
```

Here's using an arbitrary block

```
a = 3
PROC = proc { a } # gotta use something besides a local
# variable because that "class" makes us lose scope.

class A
  define_method(:go, &PROC)
end
```

or here

```
class A
end
a = 3
A.class_eval do
   define_method(:go) do
       puts a
   end
end
```

Logical And

The binary "and" operator will return the logical conjunction of its two operands. It is the same as "&&" but with a lower precedence. Example:

```
a = 1
b = 2
c = nil
puts "yay all my arguments are true" if a and b
puts "oh no, one of my argument is false" if a and c
```

Logical Or

The binary "or" operator will return the logical disjunction of its two operands. It is the same as "||" but with a lower precedence. Example:

```
a = nil
b = "foo"
c = a || b  # c is set to "foo" its the same as saying c = (a || b)
c = a or b  # c is set to nil   its the same as saying (c = a) || b which is not what you want.
```

Retrieved from "http://en.wikibooks.org/w/index.php?title=Ruby_Programming/Syntax/Operators&oldid=2274268"

Ruby Programming/Syntax/Control Structures

Control Structures

Conditional Branches

Ruby can control the execution of code using Conditional branches. A conditional Branch takes the result of a test expression and executes a block of code depending whether the test expression is true or false. If the test expression evaluates to the constant `false` or `nil`, the test is false; otherwise, it is true. Note that the number `zero` is considered true, whereas many other programming languages consider it false.

In many popular programming languages, conditional branches are statements. They can affect which code is executed, but they do not result in values themselves. In Ruby, however, conditional branches are expressions. They evaluate to values, just like other expressions do. An `if` expression, for example, not only determines whether a subordinate block of code will execute, but also results in a value itself. For instance, the following `if` expression evaluates to 3:

```
if true
  3
end
```

Ruby's conditional branches are explained below.

if expression

Examples:

```
a = 5
if a == 4
  a = 7
end
print a # prints 5 since the if-block isn't executed
```

You can also put the test expression and code block on the same line if you use `then`:

```
if a == 4 then a = 7 end
#or
if a == 4: a = 7 end
#Note that the ":" syntax for if one line blocks do not work anymore in ruby 1.9. Ternary statements st
```

This is equal to:

```
a = 5
a = 7 if a == 4
print a # prints 5 since the if-block isn't executed
```

unless expression

The unless-expression is the opposite of the if-expression, the code-block it contains will only be executed if the test expression is false.

Examples:

```
a = 5
unless a == 4
  a = 7
end
print a # prints 7 since the unless-block is executed
```

The *unless* expression is almost exactly like a negated *if* expression:

```
if !expression # is equal to using unless expression
```

The difference is that the *unless* does not permit a following *elsif*. And there is no *elsunless*.

Like the if-expression you can also write:

```
a = 5
a = 7 unless a == 4
print a # prints 7 since the unless-block is executed
```

The "one-liners" are handy when the code executed in the block is one line only.

if-elsif-else expression

The elsif (note that it's `elsif` and not `elseif`) and else blocks give you further control of your scripts by providing the option to accommodate additional tests. The `elsif` and `else` blocks are considered only if the `if` test is false. You can have any number of `elsif` blocks but only one `if` and one `else` block.

Syntax:

```
if expression
  ...code block...
elsif another expression
  ...code block...
elsif another expression
  ...code block...
else
  ...code block...
end
```

short-if expression

The "short-if" statement provides you with a space-saving way of evaluating an expression and returning a value.

The format is:

```
(condition) ? (expr if true) : (expr if false)
```

It is also known as the ternary operator and it is suggested to only use this syntax for minor tasks, such as string formatting, because of poor code readability that may result.

```
irb(main):037:0> true ? 't' : 'f'
=> "t"
irb(main):038:0> false ? 't' : 'f'
=> "f"
```

This is very useful when doing string concatenation among other things.

Example:

```
a = 5
plus_or_minus = '+'
print "The number #{a}#{plus_or_minus}1 is: " + (plus_or_minus == '+' ? (a+1).to_s : (a-1).to_s) + "."
```

case expression

An alternative to the if-elsif-else expression (above) is the case expression. Case in Ruby supports a number of syntaxes. For example, suppose we want to determine the relationship of a number (given by the variable a) to 5. We could say:

```
a = 1
case
  when a < 5 then puts "#{a} less than 5"
  when a == 5 then puts "#{a} equals 5"
  when a > 5 then puts "#{a} greater than 5"
end
```

Note that, as with if, the comparison operator is ==. The assignment operator is =. Although Ruby will accept the assignment operator:

```
  when a = 5 then puts "#{a} equals 5"   # WARNING! This code CHANGES the value of a!
```

This is not want we want! Here, we want the comparison operator.

Another equivalent syntax for case is to use ":" instead of "then":

```
when a < 5 then puts "#{a} less than 5"
```

```
when a < 5 : puts "#{a} less than 5"
```

A more concise syntax for case is to imply the comparison:

```
case a
  when 0..4 then puts "#{a} less than 5"
  when 5 then puts "#{a} equals 5"
  when 5..10 then puts "#{a} greater than 5"
  else puts "unexpected value #{a} "         #just in case "a" is bigger than 10 or negative
end
```

Note: because the ranges are explicitly stated, it is good coding practise to handle unexpected values of a. This concise syntax is perhaps most useful when we know in advance what the values to expect. For example:

```
a = "apple"
case a
  when "vanilla" then "a spice"
  when  "spinach" then "a vegetable"
  when "apple"  then "a fruit"
  else "an unexpected value"
end
```

If entered in the irb this gives:

```
=> "a fruit"
```

Other ways to use case and variations on its syntax maybe seen at Linuxtopia Ruby Programming [1] (http://www.linuxtopia.org/online_books/programming_books/ruby_tutorial/Ruby_Expressions_Case_Expressions.html)

Loops

while

The `while` statement in Ruby is very similar to `if` and to other languages' `while` (syntactically):

```
while <expression>
  <...code block...>
end
```

The code block will be executed again and again, as long as the expression evaluates to `true`.

Also, like `if` and `unless`, the following is possible:

```
<...code...> while <expression>
```

Note the following strange case works...

```
line = inf.readline while line != "what I'm looking for"
```

So if local variable `line` has no existence prior to this line, on seeing it for the first time it has the value `nil` when the loop expression is first evaluated.

until

The `until` statement is similar to the `while` statement in functionality. Unlike the `while` statement, the code block for the `until` loop will execute as long as the expression evaluates to `false`.

```
until <expression>
  <...code block...>
end
```

Keywords

return

`return value` causes the method in which it appears to exit at that point and return the value specified

Note that if no return of a value is specified in a method the value of the last value set is implicitly returned as the return value of the method.

Retrieved from "http://en.wikibooks.org/w/index.php?title=Ruby_Programming/Syntax/Control_Structures&oldid=2301990"

Ruby Programming/Syntax/Method Calls

A **method** in Ruby is a set of expressions that returns a value. Other languages sometimes refer to this as a function. A method may be defined as a part of a class or separately.

Method Calls

Methods are called using the following syntax:

```
method_name(parameter1, parameter2,…)
```

If the method has no parameters the parentheses can usually be omitted as in the following:

```
method_name
```

If you don't have code that needs to use method result immediately, Ruby allows to specify parameters omitting parentheses:

```
results = method_name parameter1, parameter2           # calling method, not using parentheses

# You need to use parentheses if you want to work with the result immediately.
# e.g., if a method returns an array and we want to reverse element order:
results = method_name(parameter1, parameter2).reverse
```

Method Definitions

Methods are defined using the keyword `def` followed by the *method name*. Method parameters are specified between parentheses following the method name. The *method body* is enclosed by this definition on the top and the word `end` on the bottom. By convention method names that consist of multiple words have each word separated by an underscore.

Example:

```
def output_something(value)
  puts value
end
```

Return Values

Methods return the value of the last statement executed. The following code returns the value `x+y`.

```ruby
def calculate_value(x,y)
  x + y
end
```

An explicit return statement can also be used to return from function with a value, prior to the end of the function declaration. This is useful when you want to terminate a loop or return from a function as the result of a conditional expression.

Note, if you use "return" within a block, you actually will jump out from the function, probably not what you want. To terminate block, use **break**. You can pass a value to break which will be returned as the result of the block:

```ruby
six = (1..10).each {|i| break i if i > 5}
```

In this case, six will have the value 6.

Default Values

A default parameter value can be specified during method definition to replace the value of a parameter if it is not passed into the method.

```ruby
def some_method(value='default', arr=[])
  puts value
  puts arr.length
end

some_method('something')
```

The method call above will output:

```
something
0
```

The following is a syntax error in Ruby 1.8

```ruby
def foo( i = 7, j ) # Syntax error in Ruby 1.8.7 Unexpected ')', expecting '='
  return i + j
end
```

The above code will work in 1.9.2 and will be logically equivalent to the snippet below

```ruby
def foo( j, i = 7)
  return i + j
```

```
end
```

Variable Length Argument List, Asterisk Operator

The last parameter of a method may be preceded by an asterisk(*), which is sometimes called the 'splat' operator. This indicates that more parameters may be passed to the function. Those parameters are collected up and an array is created.

```
def calculate_value(x,y,*otherValues)
  puts otherValues
end

calculate_value(1,2,'a','b','c')
```

In the example above the output would be ['a', 'b', 'c'].

The asterisk operator may also precede an Array argument in a method call. In this case the Array will be expanded and the values passed in as if they were separated by commas.

```
arr = ['a','b','c']
calculate_value(*arr)
```

has the same result as:

```
calculate_value('a','b','c')
```

Another technique that Ruby allows is to give a Hash when invoking a function, and that gives you best of all worlds: named parameters, and variable argument length.

```
def accepts_hash( var )
  print "got: ", var.inspect # will print out what it received
end

accepts_hash :arg1 => 'giving arg1', :argN => 'giving argN'
# => got: {:argN=>"giving argN", :arg1=>"giving arg1"}
```

You see, the arguments for accepts_hash got rolled up into one hash variable. This technique is heavily used in the Ruby On Rails API.

Also note the missing parenthesis around the arguments for the accepts_hash function call, and notice that there is no { } Hash declaration syntax around the :arg1 => '...' code, either. The above code is equivalent to the more verbose:

```
accepts_hash( :arg1 => 'giving arg1', :argN => 'giving argN' )  # argument list enclosed in parens
accepts_hash( { :arg1 => 'giving arg1', :argN => 'giving argN' } ) # hash is explicitly created
```

Now, if you are going to pass a code block to function, you need parentheses.

```
accepts_hash( :arg1 => 'giving arg1', :argN => 'giving argN' )  { |s| puts s }
accepts_hash( { :arg1 => 'giving arg1', :argN => 'giving argN' } )  { |s| puts s }
# second line is more verbose, hash explicitly created, but essentially the same as above
```

The Ampersand Operator

Much like the asterisk, the ampersand(&) may precede the last parameter of a function declaration. This indicates that the function expects a code block to be passed in. A Proc object will be created and assigned to the parameter containing the block passed in.

Also similar to the ampersand operator, a Proc object preceded by an ampersand during a method call will be replaced by the block that it contains. `Yield` may then be used.

```
def method_call
  yield
end

method_call(&someBlock)
```

Understanding blocks, Procs and methods

Introduction

Ruby provides the programmer with a set of very powerful features borrowed from the domain of functional programming, namely closures, higher-order functions and first-class functions [1]. These features are implemented in Ruby by means of code blocks, Proc objects and methods (that are also objects) - concepts that are closely related and yet differ in subtle ways. In fact I found myself quite confused about this topic, having difficulty understanding the difference between blocks, procs and methods and unsure about the best practices of using them. Additionally, having some background in Lisp and years of Perl experience, I was unsure of how the Ruby concepts map to similar idioms from other programming languages, like Lisp's functions and Perl's subroutines. Sifting through hundreds of newsgroup posts, I saw that I'm not the only one with this problem, and in fact quite a lot of "Ruby Nubies" struggle with the same ideas.

In this article I lay out my understanding of this facet of Ruby, which comes as a result of extensive research of Ruby books, documentation and comp.lang.ruby, in sincere hope that other people will find it useful as well.

Procs

Shamelessly ripping from the Ruby documentation, Procs are defined as follows: Proc objects are blocks of code that have been bound to a set of local variables. Once bound, the code may be called in different contexts and still access those variables.

A useful example is also provided:

```ruby
def gen_times(factor)
    return Proc.new {|n| n*factor }
end

times3 = gen_times(3)       # 'factor' is replaced with 3
times5 = gen_times(5)

times3.call(12)               #=> 36
times5.call(5)                #=> 25
times3.call(times5.call(4))   #=> 60
```

Procs play the role of functions in Ruby. It is more accurate to call them function objects, since like everything in Ruby they are objects. Such objects have a name in the folklore - functors. A functor is defined as an object to be invoked or called as if it were an ordinary function, usually with the same syntax, which is exactly what a Proc is.

From the example and the definition above, it is obvious that Ruby Procs can also act as closures. On Wikipedia, a closure is defined as a function that refers to free variables in its lexical context. Note how closely it maps to the Ruby definition blocks of code that have been bound to a set of local variables.

More on Procs

Procs in Ruby are first-class objects, since they can be created during runtime, stored in data structures, passed as arguments to other functions and returned as the value of other functions. Actually, the gen_times example demonstrates all of these criteria, except for “passed as arguments to other functions”. This one can be presented as follows:

```ruby
def foo (a, b)
    a.call(b)
end

putser = Proc.new {|x| puts x}
foo(putser, 34)
```

There is also a shorthand notation for creating Procs - the Kernel method lambda [2] (we’ll come to methods shortly, but for now assume that a Kernel method is something akin to a global function, which can be called from anywhere in the code). Using lambda the Proc object creation from the previous example can be rewritten as:

```ruby
putser = lambda {|x| puts x}
```

Actually, there are two slight differences between lambda and Proc.new. First, argument checking. The Ruby documentation for lambda states: Equivalent to Proc.new, except the resulting Proc objects check the number of parameters passed when called. Here is an example to demonstrate this:

```ruby
pnew = Proc.new {|x, y| puts x + y}
lamb = lambda {|x, y| puts x + y}

# works fine, printing 6
pnew.call(2, 4, 11)

# throws an ArgumentError
```

```
lamb.call(2, 4, 11)
```

Second, there is a difference in the way returns are handled from the Proc. A return from Proc.new returns from the enclosing method (acting just like a return from a block, more on this later):

```
def try_ret_procnew
    ret = Proc.new { return "Baaam" }
    ret.call
    "This is not reached"
end

# prints "Baaam"
puts try_ret_procnew
```

While return from lambda acts more conventionally, returning to its caller:

```
def try_ret_lambda
    ret = lambda { return "Baaam" }
    ret.call
    "This is printed"
end

# prints "This is printed"
puts try_ret_lambda
```

With this in light, I would recommend using lambda instead of Proc.new, unless the behavior of the latter is strictly required. In addition to being a whopping two characters shorter, its behavior is less surprising.

Methods

Simply put, a method is also a block of code. However, unlike Procs, methods are not bound to the local variables around them. Rather, they are bound to some object and have access to its instance variables [3]:

```
class Boogy
    def initialize
        @dix = 15
    end

    def arbo
        puts "#{@dix} ha\n"
    end
end

# initializes an instance of Boogy
b = Boogy.new

# prints "15 ha"
b.arbo
```

A useful idiom when thinking about methods is sending messages. Given a receiver - an object that has some method defined, we can send it a message - by calling the method, optionally providing some arguments. In the example above, calling arbo is akin to sending a message “arbo”, without arguments. Ruby supports the message sending idiom more directly, by including the send method in class Object (which is the parent of all objects in Ruby). So the following two lines are equivalent to the arbo method call:

```ruby
# method/message name is given as a string
b.send("arbo")

# method/message name is given as a symbol
b.send(:arbo)
```

Note that methods can also be defined in the "top-level" scope, not inside any class. For example:

```ruby
def say (something)
    puts something
end

say "Hello"
```

While it seems that say is "free-standing", it is not. When methods such as this are defined, Ruby silently tucks them into the Object class. But this doesn't really matter, and for all practical purposes say can be seen as an independent method. Which is, by the way, just what's called a "function" in some languages (like C and Perl). The following Proc is, in many ways similar:

```ruby
say = lambda {|something| puts something}

say.call("Hello")

# same effect
say["Hello"]
```

The [] construct is a synonym to call in the context of Proc [4]. Methods, however, are more versatile than procs and support a very important feature of Ruby, which I will present right after explaining what blocks are.

Blocks

Blocks are so powerfully related to Procs that it gives many newbies a headache trying to decipher how they actually differ. I will try to ease on comprehension with a (hopefully not too corny) metaphor. Blocks, as I see them, are unborn Procs. Blocks are the larval, Procs are the insects. A block does not live on its own - it prepares the code for when it will actually become alive, and only when it is bound and converted to a Proc, it starts living:

```ruby
# a naked block can't live in Ruby
# this is a compilation error !
{puts "hello"}

# now it's alive, having been converted
# to a Proc !
pr = lambda {puts "hello"}

pr.call
```

Is that it, is that what all the fuss is about, then ? No, not at all. The designer of Ruby, Matz saw that while passing Procs to methods (and other Procs) is nice and allows high-level functions and all kinds of fancy

functional stuff, there is one common case that stands high above all other cases - passing a single block of code to a method that makes something useful out of it, for example iteration. And as a very talented designer, Matz decided that it is worthwhile to emphasize this special case, and make it both simpler and more efficient.

Passing a block to a method

No doubt that any programmer who has spent at least a couple of hours with Ruby has been shown the following examples of Ruby glory (or something very similar):

```ruby
10.times do |i|
    print "#{i} "
end

numbers = [1, 2, 5, 6, 9, 21]

numbers.each do |x|
    puts "#{x} is " + (x >= 3 ? "many" : "few")
end

squares = numbers.map {|x| x * x}
```

(Note that `do |x| ... end` is equivalent to `{ |x| ... }`.)

Such code is IMHO part of what makes Ruby the clean, readable and wonderful language it is. What happens here behind the scenes is quite simple, or at least may be depicted in a very simple way. Perhaps Ruby doesn't implement it exactly the way I'm going to describe it, since there are optimization considerations surely playing their role - but it is definitely close enough to the truth to serve as a metaphor for understanding.

Whenever a block is appended to a method call, Ruby automatically converts it to a Proc object, but one without an explicit name. The method, however, has a way to access this Proc, by means of the yield statement. See the following example for clarification:

```ruby
def do_twice
    yield
    yield
end

do_twice {puts "Hola"}
```

The method do_twice is defined and called with an attached block. Although the method didn't explicitly ask for the block in its arguments list, the yield can call the block. This can be implemented in a more explicit way, using a Proc argument:

```ruby
def do_twice(what)
    what.call
    what.call
end

do_twice lambda {puts "Hola"}
```

This is equivalent to the previous example, but using blocks with yield is cleaner, and better optimized since only one block is passed to the method, for sure. Using the Proc approach, any amount of code blocks can be passed:

```
def do_twice(what1, what2, what3)
    2.times do
        what1.call
        what2.call
        what3.call
    end
end

do_twice(   lambda {print "Hola, "},
            lambda {print "querido "},
            lambda {print "amigo\n"})
```

It is important to note that many people frown at passing blocks, and prefer explicit Procs instead. Their rationale is that a block argument is implicit, and one has to look through the whole code of the method to see if there are any calls to yield there, while a Proc is explicit and can be immediately spotted in the argument list. While it's simply a matter of taste, understanding both approaches is vital.

The ampersand (&)

The ampersand operator can be used to explicitly convert between blocks and Procs in a couple of cases. It is worthy to understand how these work.

Remember how I said that although an attached block is converted to a Proc under the hood, it is not accessible as a Proc from inside the method ? Well, if an ampersand is prepended to the last argument in the argument list of a method, the block attached to this method is converted to a Proc object and gets assigned to that last argument:

```
def contrived(a, &f)
    # the block can be accessed through f
    f.call(a)

    # but yield also works !
    yield(a)
end

# this works
contrived(25) {|x| puts x}

# this raises ArgumentError, because &f
# isn't really an argument - it's only there
# to convert a block
contrived(25, lambda {|x| puts x})
```

Another (IMHO far more efficacious) use of the ampersand is the other-way conversion - converting a Proc into a block. This is very useful because many of Ruby's great built-ins, and especially the iterators, expect to receive a block as an argument, and sometimes it's much more convenient to pass them a Proc. The following example is taken right from the excellent "Programming Ruby" book by the pragmatic programmers:

```
print "(t)imes or (p)lus: "
times = gets
```

```
print "number: "
number = Integer(gets)
if times =~ /^t/
    calc = lambda {|n| n*number }
else
    calc = lambda {|n| n+number }
end
puts((1..10).collect(&calc).join(", "))
```

The collect method expects a block, but in this case it is very convenient to provide it with a Proc, since the Proc is constructed using knowledge gained from the user. The ampersand preceding calc makes sure that the Proc object calc is turned into a code block and is passed to collect as an attached block.

The ampersand also allows the implementation of a very common idiom among Ruby programmers: passing method names into iterators. Assume that I want to convert all words in an Array to upper case. I could do it like this:

```
words = %w(Jane, aara, multiko)
upcase_words = words.map {|x| x.upcase}

p upcase_words
```

This is nice, and it works, but I feel it's a little bit too verbose. The upcase method itself should be given to map, without the need for a separate block and the apparently superfluous x argument. Fortunately, as we saw before, Ruby supports the idiom of sending messages to objects, and methods can be referred to by their names, which are implemented as Ruby Symbols. For example:

```
p "Erik".send(:upcase)
```

This, quite literally, says send the message/method upcase to the object "Erik". This feature can be utilized to implement the map {|x| x.upcase} in an elegant manner, and we're going to use the ampersand for this ! As I said, when the ampersand is prepended to some Proc in a method call, it converts the Proc to a block. But what if we prepend it not to a Proc, but to another object ? Then, Ruby's implicit type conversion rules kick in, and the to_proc method is called on the object to try and make a Proc out of it. We can use this to implement to_proc for Symbol and achieve what we want:

```
class Symbol

    # A generalized conversion of a method name
    # to a proc that runs this method.
    #
    def to_proc
        lambda {|x, *args| x.send(self, *args)}
    end

end

# Voila !
words = %w(Jane, aara, multiko)
upcase_words = words.map(&:upcase)
```

Dynamic methods

You can define a method on "just one object" in Ruby.

```
a = 'b'
def a.some_method
  'within a singleton method just for a'
end
>> a.some_method
=> 'within a singleton method just for a'
```

Or you can use define_method, which preserves the scope around the definition, as well.

```
a = 'b'
a.class.send(:define_method, :some_method) { # only available to classes, unfortunately
  'within a block method'
}
a.some_method
```

Special methods

Ruby has a number of special methods that are called by the interpreter. For example:

```
class Chameleon
  alias __inspect__ inspect
  def method_missing(method, *arg)
    if (method.to_s)[0..2] == "to_"
      @identity = __inspect__.sub("Chameleon", method.to_s.sub('to_','').capitalize)
      def inspect
        @identity
      end
      self
    else
      super #method_missing overrides the default Kernel.method_missing
            #pass on anything we weren't looking for so the Chameleon stays unnoticed and uneaten ;
    end
  end
end
mrlizard = Chameleon.new
mrlizard.to_rock
```

This does something silly but `method_missing` is an important part of meta-programming in Ruby. In Ruby on Rails it is used extensively to create methods dynamically.

Another special methods is `initialize` that ruby calls whenever a class instance is created, but that belongs in the next chapter: Classes.

Conclusion

Ruby doesn't really have functions. Rather, it has two slightly different concepts - methods and Procs (which are, as we have seen, simply what other languages call function objects, or functors). Both are blocks of code - methods are bound to Objects, and Procs are bound to the local variables in scope. Their uses are quite different.

Methods are the cornerstone of object-oriented programming, and since Ruby is a pure-OO language (everything is an object), methods are inherent to the nature of Ruby. Methods are the actions Ruby objects do - the messages they receive, if you prefer the message sending idiom.

Procs make powerful functional programming paradigms possible, turning code into a first-class object of Ruby allowing to implement high-order functions. They are very close kin to Lisp's lambda forms (there's little doubt about the origin of Ruby's Proc constructor lambda)

The construct of a block may at first be confusing, but it turns out to be quite simple. A block is, as my metaphor goes, an unborn Proc - it is a Proc in an intermediate state, not bound to anything yet. I think that the simplest way to think about blocks in Ruby, without losing any comprehension, would be to think that blocks are really a form of Procs, and not a separate concept. The only time when we have to think of blocks as slightly different from Procs is the special case when they are passed as the last argument to a method which may then access them using yield.

That's about it, I guess. I know for sure that the research I conducted for this article cleared many misunderstandings I had about the concepts presented here. I hope others will learn from it as well. If you see anything you don't agree with - from glaring errors to small inaccuracies, feel free to amend the book.

Notes

[1] It seems that in the pure, theoretical interpretation what Ruby has isn't first-class functions per se. However, as this article demonstrates, Ruby is perfectly capable of fulfilling most of the requirements for first-class functions, namely that functions can be created during the execution of a program, stored in data structures, passed as arguments to other functions, and returned as the values of other functions.

[2] lambda has a synonym - proc, which is considered 'mildly deprecated' (mainly because proc and Proc.new are slightly different, which is confusing). In other words, just use lambda.

[3] These are 'instance methods'. Ruby also supports 'class methods', and 'class variables', but that is not what this article is about.

[4] Or more accurately, call and [] both refer to the same method of class Proc. Yes, Proc objects themselves have methods !

Ruby Programming/Syntax/Classes

Classes are the basic template from which object instances are created. A class is made up of a collection of variables representing internal state and methods providing behaviours that operate on that state.

Class Definition

Classes are defined in Ruby using the `class` keyword followed by a name. The name must begin with a capital letter and by convention names that contain more than one word are run together with each word capitalized and no separating characters (CamelCase). The class definition may contain method, class variable, and instance variable declarations as well as calls to methods that execute in the class context at read time, such as attr_accessor. The class declaration is terminated by the `end` keyword.

Example:

```
class MyClass
  def some_method
  end
end
```

Instance Variables

Instance variables are created for each class instance and are accessible only within that instance. They are accessed using the @ operator. Outside of the class definition, the value of an instance variable can only be read or modified via that instance's public methods.

Example:

```
class MyClass
  @one = 1
  def do_something
    @one = 2
  end
  def output
    puts @one
  end
end
instance = MyClass.new
instance.output
instance.do_something
instance.output
```

Surprisingly, this outputs:

```
nil
2
```

This happens (nil in the first output line) because *@one* defined below *class MyClass* is an instance variable belonging to the class object (note this is not the same as a class variable and could not be referred to as *@@one*), whereas *@one* defined inside the *do_something* method is an instance variable belonging to instances of *MyClass*. They are two distinct variables and the first is accessible only in a class method.

Accessor Methods

As noted in the previous section, an instance variable can only be directly accessed or modified within an instance method definition. If you want to provide access to it from outside, you need to define public accessor methods, for example

```
class MyClass
  def initialize
    @foo = 28
  end

  def foo
    return @foo
  end

  def foo=(value)
    @foo = value
  end
end

instance = MyClass.new
puts instance.foo
instance.foo = 496
puts instance.foo
```

Note that ruby provides a bit of syntactic sugar to make it look like you are getting and setting a variable directly; under the hood

```
a = instance.foo
instance.foo = b
```

are calls to the foo and foo= methods

```
a = instance.foo()
instance.foo=(b)
```

Since this is such a common use case, there is also a convenience method to autogenerate these getters and setters:

```
class MyClass
  attr_accessor :foo
```

```
  def initialize
    @foo = 28
  end
end

instance = MyClass.new
puts instance.foo
instance.foo = 496
puts instance.foo
```

does the same thing as the above program. The attr_accessor method is run at read time, when ruby is constructing the class object, and it generates the foo and foo= methods.

However, there is no requirement for the accessor methods to simply transparently access the instance variable. For example, we could ensure that all values are rounded before being stored in foo:

```
class MyClass
  def initialize
    @foo = 28
  end

  def foo
    return @foo
  end

  def foo=(value)
    @foo = value.round
  end
end

instance = MyClass.new
puts instance.foo
instance.foo = 496.2
puts instance.foo #=> 496
```

Class Variables

Class variables are accessed using the @@ operator. These variables are associated with the class hierarchy rather than any object instance of the class and are the same across all object instances. (These are similar to class "static" variables in Java or C++).

Example:

```
class MyClass
  @@value = 1
  def add_one
    @@value= @@value + 1
  end

  def value
```

```
    @@value
  end
end
instanceOne = MyClass.new
instanceTwo = MyClass.new
puts instanceOne.value
instanceOne.add_one
puts instanceOne.value
puts instanceTwo.value
```

Outputs:

```
1
2
2
```

Class Instance Variables

Classes can have instance variables. This gives each class a variable that is not shared by other classes in the inheritance chain.

```
class Employee
  class << self; attr_accessor :instances; end
  def store
    self.class.instances ||= []
    self.class.instances << self
  end
  def initialize name
    @name = name
  end
end
class Overhead < Employee; end
class Programmer < Employee; end
Overhead.new('Martin').store
Overhead.new('Roy').store
Programmer.new('Erik').store
puts Overhead.instances.size    # => 2
puts Programmer.instances.size  # => 1
```

For more details, see MF Bliki: ClassInstanceVariables (http://martinfowler.com/bliki/ClassInstanceVariable.html)

Class Methods

Class methods are declared the same way as normal methods, except that they are prefixed by `self`, or the class name, followed by a period. These methods are executed at the Class level and may be called without an object instance. They cannot access instance variables but do have access to class variables.

Example:

```ruby
class MyClass
  def self.some_method
    puts 'something'
  end
end
MyClass.some_method
```

Outputs:

```
something
```

Instantiation

An object instance is created from a class through the a process called *instantiation*. In Ruby this takes place through the Class method `new`.

Example:

```ruby
anObject = MyClass.new(parameters)
```

This function sets up the object in memory and then delegates control to the initialize function of the class if it is present. Parameters passed to the new function are passed into the `initialize` function.

```ruby
class MyClass
  def initialize(parameters)
  end
end
```

Declaring Visibility

By default, all methods in Ruby classes are public - accessible by anyone. There are, nonetheless, only two exceptions for this rule: the global methods defined under the Object class, and the initialize method for any class. Both of them are implicitly private.

If desired, the access for methods can be restricted by public, private, protected object methods.

It is interesting that these are not actually keywords, but actual methods that operate on the class, dynamically altering the visibility of the methods, and as a result, these 'keywords' influence the visibility of all following declarations until a new visibility is set or the end of the declaration-body is reached.

Private

Simple example:

```ruby
class Example
  def methodA
  end
```

```
  private # all methods that follow will be made private: not accessi

  def methodP
  end
end
```

If private is invoked without arguments, it sets access to private for all subsequent methods. It can also be invoked with named arguments.

Named private method example:

```
class Example
  def methodA
  end

  def methodP
  end

  private :methodP
end
```

Here private was invoked with an argument, altering the visibility of methodP to private.

Note for class methods (those that are declared using def ClassName.method_name), you need to use another function: private_class_method

Common usage of private_class_method is to make the "new" method (constructor) inaccessible, to force access to an object through some getter function. A typical Singleton implementation is an obvious example.

```
class SingletonLike
  private_class_method :new

  def SingletonLike.create(*args, &block)
    @@inst = new(*args, &block) unless @@inst
    return @@inst
  end
end
```

Note : another popular way to code the same declaration

```
class SingletonLike
  private_class_method :new

  def SingletonLike.create(*args, &block)
    @@inst ||= new(*args, &block)
  end
end
```

More info about the difference between C++ and Ruby private/protected: http://lylejohnson.name/blog/?p=5

One person summed up the distinctions by saying that in C++, “private” means “private to this class”, while in Ruby it means “private to this instance”. What this means, in C++ from code in class A, you can access any private method for any other object of type A. In Ruby, you can not: you can only access private methods for your instance of object, and not for any other object instance (of class A).

Ruby folks keep saying "private means you cannot specify the receiver". What they are saying, if method is private, in your code you can say:

```
class AccessPrivate
  def a
  end
  private :a # a is private method

  def accessing_private
    a              # sure!
    self.a         # nope! private methods cannot be called with an e
    other_object.a # nope, a is private, you can't get it (but if it
  end
end
```

Here, "other_object" is the "receiver" that method "a" is invoked on. For private methods, it does not work. However, that is what "protected" visibility will allow.

Public

Public is default accessibility level for class methods. I am not sure why this is specified - maybe for completeness, maybe so that you could dynamically make some method private at some point, and later - public.

In Ruby, visibility is completely dynamic. You can change method visibility at runtime!

Protected

Now, protected deserves more discussion. Those of you coming from Java (or C++) background, you know that "private" means that method visibility is restricted to the declaring class, and if method is "protected", it will be accessible for children of the class (classes that inherit from parent) or other classes in that package.

In Ruby, private visibility is similar to what protected is in Java. Private methods in Ruby are accessible from children. You can't have truly private methods in Ruby; you can't completely hide a method.

The difference between protected and private is subtle. If a method is protected, it may be called by any instance of the defining class or its subclasses. If a method is private, it may be called only within the context of the calling object---it is never possible to access another object instance's private methods directly, even if the object is of the same class as the caller. For protected methods, they are accessible from objects of the same class (or children).

So, from within an object "a1" (an instance of Class A), you can call private methods only for instance of "a1" (self). And you can not call private methods of object "a2" (that also is of class A) - they are private to a2. But you can call protected methods of object "a2" since objects a1 and a2 are both of class A.

Ruby FAQ (http://www.rubycentral.com/faq/rubyfaq-7.html) gives following example - implementing an

operator that compares one internal variable with a variable from another class (for purposes of comparing the objects):

```
def <=>(other)
  self.age <=> other.age
end
```

If age is private, this method will not work, because other.age is not accessible. If "age" is protected, this will work fine, because self and other are of same class, and can access each other's protected methods.

To think of this, protected actually reminds me of the "internal" accessibility modifier in C# or "default" accessibility in Java (when no accessibility keword is set on method or variable): method is accessible just as "public", but only for classes inside the same package.

Instance Variables

Note that object instance variables are not really private, you just can't see them. To access an instance variable, you need to create a getter and setter.

Like this (no, don't do this by hand! See below):

```
class GotAccessor
  def initialize(size)
    @size = size
  end

  def size
    @size
  end
  def size=(val)
    @size = val
  end
end

# you could the access @size variable as
# a = GotAccessor.new(5)
# x = a.size
# a.size = y
```

Luckily, we have special functions to do just that: attr_accessor, attr_reader, attr_writer. attr_accessor will give you get/set functionality, reader will give only getter and writer will give only setter.

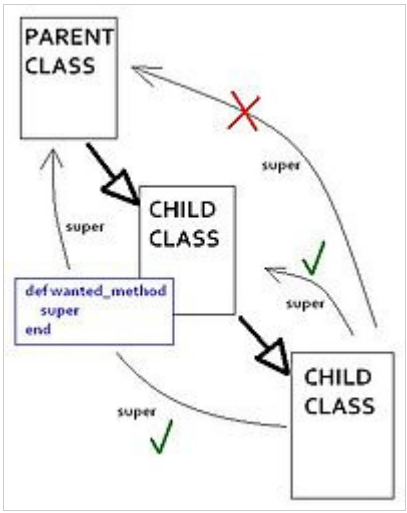
Now reduced to:

```
class GotAccessor
  def initialize(size)
    @size = size
  end

  attr_accessor :size
```

```
end

# attr_accessor generates variable @size accessor methods automatical
# a = GotAccessor.new(5)
# x = a.size
# a.size = y
```

Inheritance

A class can *inherit* functionality and variables from a *superclass*, sometimes referred to as a *parent class* or *base class*. Ruby does not support *multiple inheritance* and so a class in Ruby can have only one superclass. The syntax is as follows:



The *super* keyword only accessing the direct parents method. There is a workaround though.

```
class ParentClass
  def a_method
    puts 'b'
  end
end

class SomeClass < ParentClass  # < means inher
  def another_method
    puts 'a'
  end
end

instance = SomeClass.new
instance.another_method
instance.a_method
```

Outputs:

```
a
b
```

All non-private variables and functions are inherited by the child class from the superclass.

If your class overrides a method from parent class (superclass), you still can access the parent's method by using 'super' keyword.

```
class ParentClass
  def a_method
    puts 'b'
  end
end

class SomeClass < ParentClass
  def a_method
```

```
      super
      puts 'a'
    end
  end

  instance = SomeClass.new
  instance.a_method
```

Outputs:

```
b
a
```

(because a_method also did invoke the method from parent class).

If you have a deep inheritance line, and still want to access some parent class (superclass) methods directly, you can't. *super* only gets you a direct parent's method. But there is a workaround! When inheriting from a class, you can alias parent class method to a different name. Then you can access methods by alias.

```
  class X
    def foo
      "hello"
    end
  end

  class Y < X
    alias xFoo foo
    def foo
      xFoo + "y"
    end
  end

  class Z < Y
    def foo
      xFoo + "z"
    end
  end

  puts X.new.foo
  puts Y.new.foo
  puts Z.new.foo
```

Outputs

```
hello
helloy
helloz
```

Mixing in Modules

First, you need to read up on modules. Modules are a way of grouping together some functions and variables and classes, somewhat like classes, but more like namespaces. So a module is not really a class. You can't instantiate a Module, and thus it does not have *self*.

This trait, however, allows us to include the module into a class. Mix it in, so to speak.

```
module A
  def a1
    puts 'a1 is called'
  end
end

module B
  def b1
    puts 'b1 is called'
  end
end

module C
  def c1
    puts 'c1 is called'
  end
end

class Test
  include A
  include B
  include C

  def display
    puts 'Modules are included'
  end
end

object=Test.new
object.display
object.a1
object.b1
object.c1
```

Outputs:

```
Modules are included
a1 is called
b1 is called
c1 is called
```

The code shows Multiple Inheritance using modules.

Ruby Class Meta-Model

In keeping with the Ruby principle that everything is an object, classes are themselves instances of the class Class. They are stored in constants under the scope of the module in which they are declared. A call to a method on an object instance is delegated to a variable inside the object that contains a reference to the class of that object. The method implementation exists on the Class instance object itself. Class methods are implemented on meta-classes that are linked to the existing class instance objects in the same way that those classes instances are linked to them. These meta-classes are hidden from most Ruby functions.