Ruby (and Rails) Programming

PDF generated using the open source mwlib toolkit. See http://code.pediapress.com/ for more information.
PDF generated at: Fri, 09 Apr 2010 16:44:15 UTC

Contents

Articles

Ruby Programming 1
Ruby Programming/Overview 6
Ruby Programming/Installing Ruby 8
Ruby Programming/Ruby editors 10
Ruby Programming/Notation conventions 12
Ruby Programming/Interactive Ruby 13
Ruby Programming/Mailing List FAQ 14
Ruby Programming/Hello world 15
Ruby Programming/Strings 18
Ruby Programming/Alternate quotes 20
Ruby Programming/Here documents 21
Ruby Programming/ASCII 25
Ruby Programming/Introduction to objects 29
Ruby Programming/Ruby basics 32
Ruby Programming/Data types 37
Ruby Programming/Writing methods 43
Ruby Programming/Classes and objects 44
Ruby Programming/Exceptions 47
Ruby Programming/Unit testing 47
Ruby Programming/RubyDoc 52
Ruby Programming/Syntax 56
Ruby Programming/Syntax/Lexicology 58
Ruby Programming/Syntax/Variables and Constants 59
Ruby Programming/Syntax/Literals 62
Ruby Programming/Syntax/Operators 68
Ruby Programming/Syntax/Control Structures 71
Ruby Programming/Syntax/Method Calls 75
Ruby Programming/Syntax/Classes 86
Ruby Programming/Reference 95
Ruby Programming/Reference/Predefined Variables 96
Ruby Programming/Reference/Predefined Classes 98
Ruby Programming/Reference/Objects 99
Ruby Programming/Reference/Objects/Array 99
Ruby Programming/Object/NilClass 100

Ruby Programming/Reference/Objects/Exception 100
Ruby Programming/Reference/Objects/FalseClass 101
Ruby Programming/Reference/Objects/IO/File/File::Stat 101
Ruby Programming/Reference/Objects/Numeric 101
Ruby Programming/Reference/Objects/Numeric/Integer 103
Ruby Programming/Reference/Objects/Regexp 106
Ruby Programming/Reference/Objects/String 108
Ruby Programming/Reference/Objects/Symbol 108
Ruby Programming/Reference/Objects/Time 108
Ruby Programming/Reference/Objects/TrueClass 110
Ruby Programming/Reference/Built-in Modules 110
Ruby Programming/Built-in Modules 111
Ruby Programming/GUI Toolkit Modules/Tk 111
Ruby Programming/GUI Toolkit Modules/GTK2 111
Ruby Programming/GUI Toolkit Modules/Qt4 112
Ruby Programming/XML Processing/REXML 112
Ruby on Rails/Print version 113

References

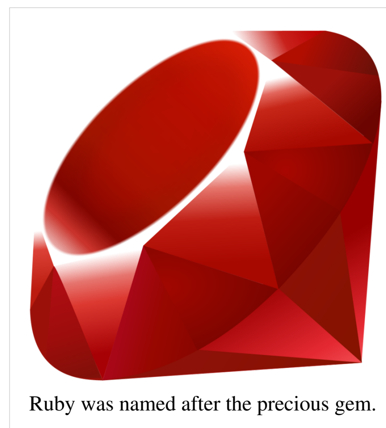
Article Sources and Contributors 169
Image Sources, Licenses and Contributors 171

Article Licenses

License 172

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.


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 final section, Intermediate Ruby covers a selection of slightly more advanced topics. Each section is designed to be self contained. Finally, the Ruby language section is organized like a reference to the language.

Table of Contents

Getting started

/Overview/

/Installing Ruby/

/Ruby editors/

/Notation conventions/

/Interactive Ruby/

/Mailing List FAQ/

Basic Ruby

/Hello world/

/Strings/

/Alternate quotes/

/Here documents/

/ASCII/

/Introduction to objects/

/Ruby basics/

/Data types/ -- numbers, strings, hashes and arrays

/Writing methods/

/Classes and objects/

/Exceptions/

Intermediate Ruby

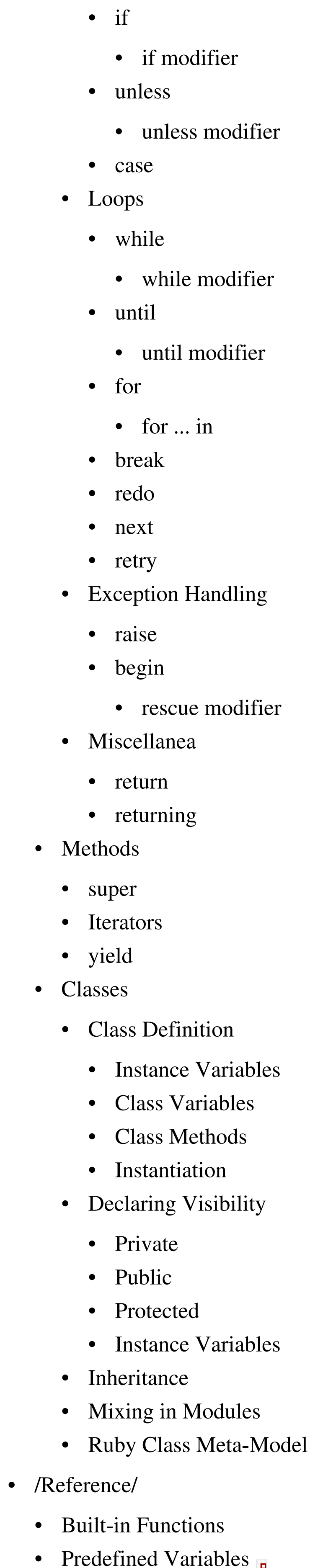
/Unit testing/

/RubyDoc/

/Rake/

Ruby reference

- /Syntax/
  - Lexicology
    - Identifiers
    - Comments
    - Embedded Documentation
    - Reserved Words
    - Expressions
  - Variables and Constants
    - Local Variables
    - Instance Variables
    - Class Variables
    - Global Variables
    - Constants
    - Pseudo Variables
    - Pre-defined Variables
  - Literals
    - Numerics
    - Strings
      - Interpolation
      - Backslash Notation
      - The % Notation
    - ~~Command Expansion~~
    - ~~Regular Expressions~~
    - Arrays
    - Hashes
    - Ranges
    - Symbols
  - Operators
    - Assignment
      - Self Assignment
      - Multiple Assignment
      - Conditional Assignment
    - Scope
    - and
    - or
    - not
  - Control Structures
    - Conditional Branches



            - if
                - if modifier
            - unless
                - unless modifier
            - case
        - Loops
            - while
                - while modifier
            - until
                - until modifier
            - for
                - for ... in
            - break
            - redo
            - next
            - retry
        - Exception Handling
            - raise
            - begin
                - rescue modifier
        - Miscellanea
            - return
            - returning
    - Methods
        - 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
- /Reference/
    - Built-in Functions
    - Predefined Variables

- Predefined Classes
  - Object
    - Array
    - Class
    - Exception
    - FalseClass
    - IO
      - File
        - File::Stat
    - Method
    - Module
      - Class
    - NilClass
    - Numeric
      - Integer
        - Bignum
        - Fixnum
      - Float
    - Range
    - Regexp
    - String
    - Struct
      - Struct::Tms
    - Symbol
    - Time
    - TrueClass

Modules

- /Built-in Modules/
- /Database Interface Modules/
- /GUI Toolkit Modules/
  - Tk
  - GTK2
  - Qt4
- /XML Processing/
  - REXML

External links

- Ruby homepage [1]
- Ruby Documentation Homepage [2]
- Ruby-Talk: The Ruby Mailing List [3]
- Ruby's Nitro/Og Web Toolkit [4]
- Ruby on Rails [5]
- Ruby on Rails plugin directory [6]
- RubyForge: The Repository for Open-source Ruby Projects [7]
- Ruby Wizards Discussion Forum [8]

Learning Ruby

- A Ruby Tutorial that Anyone can Edit [9]
- Learning Ruby [10] A free tool to find and learn Ruby concepts

Books

- The Ruby Programming Language [11] by David Flanagan, Yukihiro Matsumoto aka "Matz", the creator of Ruby.
- Programming Ruby 3 [12] (aka "Pickaxe") - this 2008 version covers Ruby 1.9
- Programming Ruby (a.k.a. 'Pickaxe Book') 1st edition online [13]
- by Example [14]
- Ruby Study Notes [15]
- Why's (Poignant) Guide To Ruby [16]
- Humble Little Ruby Book [17]

Quick Reference

- Ruby Quick Reference (some of more obscure expressions are explained) [18]
- Ruby Cheat Sheets (a list of some different Ruby cheat sheets) [19]

References

[1] http://www.ruby-lang.org/
[2] http://www.ruby-doc.org/
[3] http://ruby-talk.org/ruby/ruby-talk/index.shtml
[4] http://www.nitroproject.org/
[5] http://www.rubyonrails.com/
[6] http://www.railslodge.com/
[7] http://www.rubyforge.org/
[8] http://www.rubywizards.com/
[9] http://www.meshplex.org/wiki/Ruby/Ruby_on_Rails_programming_tutorials
[10] http://www.yoyobrain.com/cardboxes/preview/103
[11] http://www.oreilly.com/catalog/9780596516178/
[12] http://pragprog.com/titles/ruby3
[13] http://www.rubycentral.com/book/
[14] http://www.oreilly.com/catalog/9781593271480/Ruby
[15] http://rubylearning.com/download/downloads.html
[16] http://poignantguide.net/ruby/
[17] http://www.humblelittlerubybook.com/book/
[18] http://www.zenspider.com/Languages/Ruby/QuickRef.html
[19] http://www.rubyinside.com/ruby-cheat-sheet-734.html

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 [1]. 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.[2]

Features

Ruby combines features from Perl, Smalltalk, Eiffel, Ada, Lisp, and Python.[2]

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 hiearchy 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 hiearchies.

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 who's 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] Bruce Stewart (November 29, 2001). An Interview with the Creator of Ruby (http://www.linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html). O'Reilly. Retrieved on 2006-09-11.
[2] Download Ruby (http://www.ruby-lang.org/en/downloads/). Retrieved on 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 [1].

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 Fink.
- Install Ruby using MacPorts.

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.[2] Otherwise, you can install Ruby using the command-line tool yum.

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 [1].
- Download and run One-Click Ruby Installer [3].
- 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.

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] http://www.ruby-lang.org/en/downloads/
[2] yum (http://fedoraproject.org/wiki/Tools/yum). Fedora Wiki. Retrieved on 2006-09-13.
[3] http://rubyforge.org/projects/rubyinstaller

Ruby Programming/Ruby editors

Although you can write Ruby programs with any text editor, some text editors have additional features to aid the Ruby programmer. The most common is syntax highlighting. The following editors have built-in support for Ruby. They are listed alphabetically in each section, not by preference.

Operating systems

Cross-platform

- 3rdRail [1]
- ActiveState Komodo and its stripped down cousin Komodo Edit (free) supports Ruby syntax highlighting and auto-complete [2]
- Arachno [3]
- Eclipse with the Dynamic Languages Toolkit Plugin [4]
- Eclipse with the RDT Plugin [5]
- Emacs
- FreeRIDE [6]
- IDEA with the Ruby Plugin [7]
- jEdit with the Ruby Editor Plugin [8]
- NetBeans [9]
- RubyMine
- SciTE
- SlickEdit [10]
- Vim — Learn Vim here on Wikibooks
- XEmacs
- Geany [11]

Mac OS X

- BBEdit
- TextWrangler
- TextMate
- Coda

Linux/Unix

- Kate, which is part of KDE
- Scribes [12]
- Gedit
- red car editor [13] written in ruby

All of the editors listed in the cross-platform section will run on Linux.

Windows

- e - A Textmate like editor for Windows
- Intype [14] - A small, fast, and flexible code editor (Alpha 0.3.x)
- Notepad++
- TextPad with the Ruby syntax definition files [15].
- Tse - syntax highlighting, compile your Ruby programs while in the editor
- UltraEdit with the UltraEdit extensions [16]
- Ruby In Steel [17] - Ruby and Rails editing and debugging for Visual Studio 2008.
- EditPad Pro [18] - **Pro** (non-free) version includes built-in syntax highlighting (w/ spellcheck) for Ruby.

References

[1] http://www.codegear.com/products/3rdrail
[2] http://www.activestate.com/Products/Komodo/
[3] http://www.ruby-ide.com/
[4] http://www.eclipse.org/dltk/
[5] http://rubyeclipse.sourceforge.net/
[6] http://freeride.rubyforge.org
[7] http://www.jetbrains.net/confluence/display/RUBYDEV/Home
[8] http://rubyjedit.org/
[9] http://www.netbeans.org/products/ruby/
[10] http://www.slickedit.com/
[11] http://www.geany.org
[12] http://scribes.sourceforge.net/
[13] http://redcareditor.com/
[14] http://intype.info
[15] http://www.textpad.com/add-ons/synn2t.html
[16] http://rubygarden.org/ruby/ruby?UltraEditExtensions
[17] http://www.sapphiresteel.com
[18] http://www.editpadpro.com

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.

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

Ruby Programming/Mailing List FAQ

Etiquette

There is a list of Best Practices[1] .

What is the Best editor?

Several editors exist for Ruby.

Commercial: The editor of preference on OS X is Textmate. RubyMine has also received good reviewed[2]

Free: NetBeans offers a version with Ruby support. RadRails is a port of eclipse to support Ruby syntax. Eclipse has a plugin DLTK that offers ruby support[3] . On windows for rails projects there is RoRed.

[1] http://blog.rubybestpractices.com/posts/jamesbritt/and_your_Mom_too.html
[2] http://www.rubyinside.com/rubymine-1-0-ruby-ide-1818.html
[3] http://www.infoq.com/news/2007/08/eclipse-dltk-09

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 [1], 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"
```

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. 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}/

Ruby Programming/Here documents

For creating multiple-line strings, Ruby supports here documents, 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

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 a 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
```

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	[[w:Exclamation mark	]]
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	'	
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	=	
011 1110	076	62	3E	>	
011 1111	077	63	3F	?	

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

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
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	}
111 1110	176	126	7E	~

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.

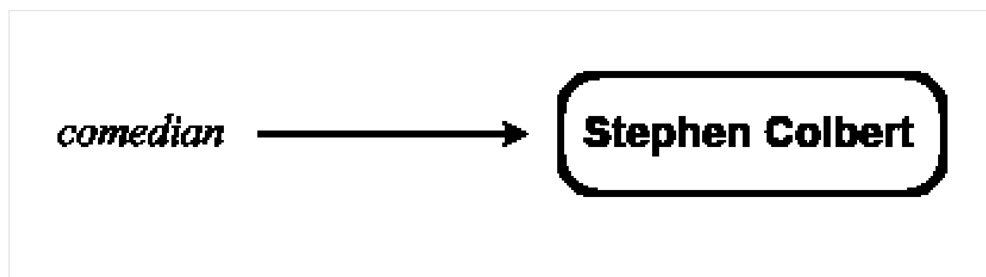
If you don't come from Smalltalk background, it might make more sense to rephrase these rules as follows:

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.

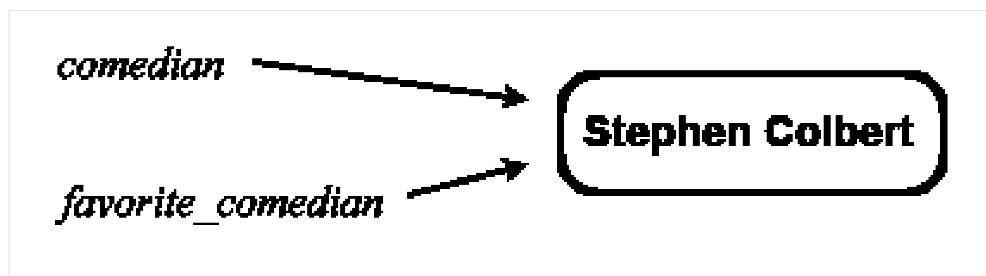
```
$ irb --simple-prompt
>> comedian = "Stephen Colbert"
=> "Stephen Colbert"
```



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 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"
```

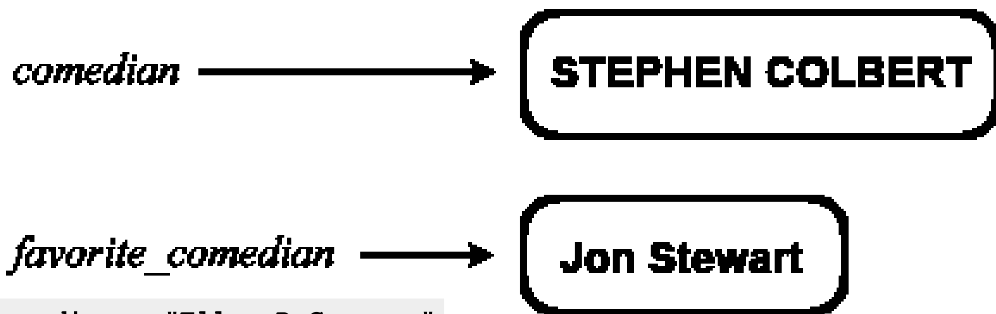
Since the original object still contains the text "STEPHEN COLBERT", you might wonder where the new String object, with the lowercase text, went to. Well, after irb printed out its contents, it can no longer be accessed since we did not assign a variable to keep track of it.It's essentially gone, and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

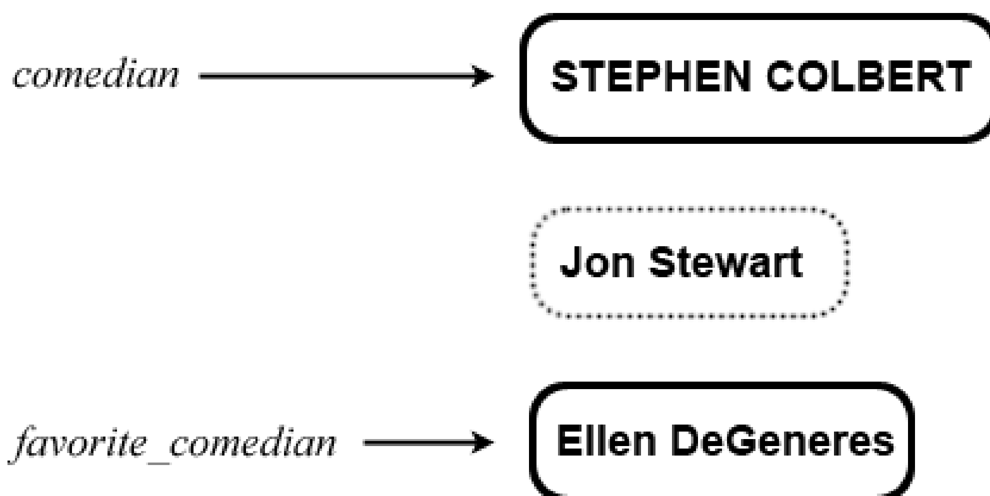
Now, each variable points to a different object.



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.



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 infact 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.

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, :linux => "fun for coders."}
puts "Stringy string McString!".class
puts 1.class
puts nil.class
puts h.class
puts :symbol.class
```

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.
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 hell 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
=> -606662268
irb> "live long and prosper".object_id
=> -606666538

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 Jedi"}
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 Jedi"}
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 turned Jedi")
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!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
```

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 100..."
100.times {|number| puts number}
5.times {puts "Guess what?"}
puts "I'm done!"
```

This will print out the numbers 1 through 100, 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 brackets 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
myself: " + msg
   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
method is defined
           #This will be covered in a later section when dealing with
Classes
           if(oVal == @value)
               return true
           else
               return false
           end
       end
```

end

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:

```
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:

```
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:

```
# 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 it allow us to define methods without the `self.` prefix.

Ruby Programming/Exceptions

There is no exceptions chapter at present. Instead, here is a link to a chapter about exceptions from **Yukihiro Matsumoto**'s book, *Programming Ruby: The Pragmatic Programmer's Guide* [1] More detail and simpler examples about exceptions, by **Satish Talim**, maybe found in a tutorial at RubyLearning.com [2]

References

[1] http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut_exceptions.html
[2] http://rubylearning.com/satishtalim/ruby_exceptions.html

Ruby Programming/Unit testing

Unit testing is a great way to catch minor errors early in the development process, if you dedicate time to writing appropriate and useful tests. As in other languages, Ruby provides a framework for setting up, organizing, and running tests called Test::Unit.

Specifically, Test::Unit provides three basic functionalities:

1. A way to define basic pass/fail tests.
2. A way to gather like tests together and run them as a group.
3. Tools for running single tests or whole groups of tests.

A Simple Introduction

First create a new class.

```
class SimpleNumber

  def initialize( num )
    raise unless num.is_a?(Numeric)
    @x = num
  end

  def add( y )
    @x + y
  end

  def multiply( y )
    @x * y
  end

end
```

Let's start with an example to test the SimpleNumber class.

```
 require "simpleNumber"
 require "test/unit"

 class TestSimpleNumber < Test::Unit::TestCase
```

```
  def test_simple
     assert_equal(4, SimpleNumber.new(2).add(2) )
     assert_equal(6, SimpleNumber.new(2).multiply(3) )
   end

end
```

Which produces

```
>> ruby tc_simpleNumber.rb
Loaded suite tc_simpleNumber
Started
.
Finished in 0.002695 seconds.

1 tests, 2 assertions, 0 failures, 0 errors
```

So what happened here? We defined a class *TestSimpleNumber* which inherited from *Test::Unit::TestCase*. In *TestSimpleNumber* we defined a member function called *test_simple*. That member function contains a number of simple *assertions* which exercise my class. When we run that class (note I haven't put any sort of wrapper code around it -- it's just a class definition), the tests are automatically run, and we're informed that we've run 1 test and 2 assertions.

Let's try a more complicated example.

```
require "simpleNumber"
require "test/unit"

class TestSimpleNumber < Test::Unit::TestCase

  def test_simple
    assert_equal(4, SimpleNumber.new(2).add(2) )
    assert_equal(4, SimpleNumber.new(2).multiply(2) )
  end

  def test_typecheck
    assert_raise( RuntimeError ) { SimpleNumber.new('a') }
  end

  def test_failure
    assert_equal(3, SimpleNumber.new(2).add(2), "Adding doesn't work" )
  end

end
```

```
>> ruby tc_simpleNumber2.rb
Loaded suite tc_simpleNumber2
Started
F..
```

```
Finished in 0.038617 seconds.

  1) Failure:
test_failure(TestSimpleNumber) [tc_simpleNumber2.rb:16]:
Adding doesn't work.
<3> expected but was
<4>.

3 tests, 4 assertions, 1 failures, 0 errors
```

Now there are three tests (three member functions) in the class. The function *test_typecheck* uses *assert_raise* to check for an exception. The function *test_failure* is set up to fail, which the Ruby output happily points out, not only telling us which test failed, but how it failed (expected <3> but was <4>). On this assertion, we've also added an final parameters which is a custom error message. It's strictly optional but can be helpful for debugging. All of the assertions include their own error messages which are usually sufficient for simple debugging.

Available Assertions

Test::Unit provides a rich set of assertions, which are documented thoroughly at Ruby-Doc [1]. Here's a brief synopsis (assertions and their negative are grouped together. The text description is usually for the first one listed -- the names should make some logical sense):

assert(boolean, [message])	True if *boolean*
assert_equal(expected, actual, [message]) assert_not_equal(expected, actual, [message])	True if *expected == actual*
assert_match(pattern, string, [message]) assert_no_match(pattern, string, [message])	True if *string =~ pattern*
assert_nil(object, [message]) assert_not_nil(object, [message])	True if *object == nil*
assert_in_delta(expected_float, actual_float, delta, [message])	True if *(actual_float - expected_float).abs <= delta*
assert_instance_of(class, object, [message])	True if *object.class == class*
assert_kind_of(class, object, [message])	True if *object.kind_of?(class)*
assert_same(expected, actual, [message]) assert_not_same(expected, actual, [message])	True if *actual.equal?(expected).*
assert_raise(Exception,...) {block} assert_nothing_raised(Exception,...) {block}	True if the block raises (or doesn't) one of the listed exceptions.
assert_throws(expected_symbol, [message]) {block} assert_nothing_thrown([message]) {block}	True if the block throws (or doesn't) the expected_symbol.
assert_respond_to(object, method, [message])	True if the object can respond to the given method.
assert_send(send_array, [message])	True if the method sent to the object with the given arguments return true.
assert_operator(object1, operator, object2, [message])	Compares the two objects with the given operator, passes if *true*

Structuring and Organizing Tests

Tests for a particular unit of code are grouped together into a *test case*, which is a subclass of Test::Unit::TestCase. Assertions are gathered in *tests*, member functions for the test case whose names start with `test_`. When the test case is executed or required, Test::Unit will iterate through all of the tests (finding all of the member functions which start with *test_* using reflection) in the test case, and provide the appropriate feedback.

Test case classes can be gathered together into *test suites* which are Ruby files which require other test cases:

```
# File: ts_allTheTests.rb
require 'test/unit'
require 'testOne'
require 'testTwo'
require 'testThree'
```

In this way, related test cases can be naturally grouped. Further, test suites can contain other test suites, allowing the construction of a hierarchy of tests.

This structure provides relatively fine-grained control over testing. Individual tests can be run from a test case (see below), a full test case can be run stand-alone, a test suite containing multiple cases can be run, or a suite of suites can run, spanning many test cases.

Naming Conventions

The author of Test::Unit, Nathaniel Talbott, suggests starting the names of test cases with `tc_` and the names of test suites with `ts_`

Running Specific Tests

It's possible to run just one (or more) tests out of a full test case:

```
>> ruby -w tc_simpleNumber2.rb --name test_typecheck
Loaded suite tc_simpleNumber2
Started
.
Finished in 0.003401 seconds.

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

It is also possible to run all tests whose names match a given pattern:

```
>> ruby -w tc_simpleNumber2.rb --name /test_type.*/
Loaded suite tc_simpleNumber2
Started
.
Finished in 0.003401 seconds.

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

Setup and Teardown

There are many cases where a small bit of code needs to be run before and/or after each test. Test::Unit provides the `setup` and `teardown` member functions, which are run before and after every test (member function).

```
require "simpleNumber"
require "test/unit"

class TestSimpleNumber < Test::Unit::TestCase

  def setup
    @num = SimpleNumber.new(2)
  end

  def teardown
    ## Nothing really
  end

  def test_simple
    assert_equal(4, @num.add(2) )
  end

  def test_simple2
    assert_equal(4, @num.multiply(2) )
  end

end
```

```
>> ruby tc_simpleNumber3.rb
Loaded suite tc_simpleNumber3
Started
..
Finished in 0.00517 seconds.

2 tests, 2 assertions, 0 failures, 0 errors
```

External Links

- Test::Unit documentation [1] at Ruby-Doc [2]

References

[1] http://www.ruby-doc.org/stdlib/libdoc/test/unit/rdoc/classes/Test/Unit.html

Ruby Programming/RubyDoc

The standard Ruby library provides a tool for generating documentation from self documenting code called RubyDoc [1] or RDoc for short. Though it's part of the standard library, RDoc isn't a file you `require` to use. Instead it consists of a command-line program called `rdoc` and a library for parsing specially-formatted comments out of Ruby and C code.

A Simple Example

Here are the contents of the file `simpleNumber.rb` which contains a class `SimpleNumber`

```
# This file is called simpleNumber.rb
# It contains the class SimpleNumber

class SimpleNumber

  def initialize( num )
    raise if !num.is_a?(Numeric)
    @x = num
  end

  def add( y )
    @x + y
  end

  def multiply( y )
    @x * y
  end

end
```

Just to get going, run `rdoc` on this file

```
>> rdoc simpleNumber.rb

                    simpleNumber.rb: c...
Generating HTML...

Files:   1
Classes: 1
Modules: 0
Methods: 3
Elapsed: 0.426s
```

A quick look around reveals `rdoc` has created a new directory called `doc/`. It contains quite a few files (given such a simple class!)

```
>> ls doc/
classes      files                 fr_file_index.html    index.html
created.rid  fr_class_index.html   fr_method_index.html  rdoc-style.css
```

Open the file `doc/index.html` in a web browser and find

What has RDoc done? First, it has scanned the given file (shown in the upper left pane). The middle pane lists all of the classes in `simpleNumber.rb`, and the right panel gives all of the methods in SimpleNumber. In this case the lower panel is showing the summary information for the file `simpleNumber.rb` which has been extracted from the top-level comment in the file. Once we select either a class or a function it will show more particular information.

Handling Multiple Files

Any directories included on the command line will be searched for Ruby files (with an `.rb` or `.rbw` extension) and C files (with a `.c` extension). rdoc will search recursively, which makes it convenient for documenting large trees of code. If no source files are specified, RubyDoc will start searching with the current directory. The output is always generated in a directory `doc/` under the current location.

Any files which aren't recognized as Ruby or C are not parsed by default, but can be included as command-line arguments to rdoc. In this case they are treated as SimpleMarkup (see below) -- generally speaking they will just be browsable as plain text files in the HTML documention.

Formatting for RDoc

RDoc uses a formatting system called SimpleMarkup for its markup purposes. SimpleMarkup is a simplified text-based markup system, much like many wiki markup languages. Here's a brief overview:

- Consecutive lines aligned on the left margin (either column 0 in a text file, or after the comment character and whitespace in a source file) are considered a paragraph. An empty line starts a new paragraph.
- If a line starts with "*", "-" or "<a digit>" the following information is a list item, either unnumbered or numbered. All subsequent lines should be indented to match the text on the first line of the list item. Here's an example (from the RDoc documentation):

```
* this is a list with three paragraphs in
  the first item. This is the first paragraph.

  And this is the second paragraph.

  1. This is an indented, numbered list.
  2. This is the second item in that list

  This is the third conventional paragraph in the
  first list item.

* This is the second item in the original list
```

- The lists can be labeled. Each label must be inside square brackets or followed by two colons:

```
    [cat]  a small furry mammal
           that seems to sleep a lot

    [ant]  a little insect that is known
           to enjoy picnics
```

```
    cat::  a small furry mammal
           that seems to sleep a lot
```

```
    ant::  a little insect that is known
           to enjoy picnics
```

- Any line which starts to the right of the current left margin is considered verbatim text (like program code, for example)
- Any line starting with one or more equals signs is a heading. One equals sign is a top-level header, two equals signs is a second-level heading, etc.
- Three or more dashes on a line give a horizontal line.
- Basic text markup:
  - Tags can be used, including asterisks for ***bold text***, underscores for _*emphasis*_ text, and plus signs for `+code+` text.
  - Alternately, HTML tags can be used.

Including Diagrams

If the program `dot` from the Graphviz [2] is available, RDoc will generate class hierarchy diagrams as part of the HTML documentation. This behavior can be enabled with the command line option `--diagram` or `-d`.

Using RDoc to Generate Usage Output

RDoc can be used to generate a command-line usage string (paragraph) from a file's comments. This is done with the class method `RDoc::usage`. Here's an example:

```
# TestRDocUsage: A useless file
#
#  Usage:  ruby testRDocUsage.rb  [options]
#

require 'rdoc/usage'

RDoc::usage
```

Which when run:

```
>> ruby testRdocUsage.rb
TestRDocUsage: A useless file

 Usage:  ruby testRDocUsage.rb  [options]
```

Not particularly interesting. `RDoc::usage` can take two kinds of command line options. If the first argument is an integer, this value is used as the exit code for the application (yes, `RDoc::usage` terminates the application). Any further (string) arguments call out sections of the introductory comment which should be used. Here's a slightly more complicated version of the above example:

```
#= Overview
# TestRDocUsage: A useless file
#
#= Usage
#
#  Usage:  ruby testRDocUsage.rb  [options]
```

```
#

require 'rdoc/usage'

RDoc::usage('Usage')
```

This time with section headers. Running the script gives

```
>> ruby testRdocUsage.rb

USAGE
=====
  Usage:  ruby testRDocUsage.rb  [options]
```

Where only the 'Usage' section (called out as an argument to `RDoc::usage`) is included.

Other Interesting Command-Line Options

Here are a selection of other interesting rdoc command-line options which haven't been covered above:

-a, --all

Parse all class methods, not just public methods.

-x, --exclude pattern

Exclude files/directories matching *pattern* from search. Files included explicitly on the command line are never excluded.

External Links

- RDoc documentation [1] (generated using RDoc!) at Ruby-Doc [2]

Questions

- How do you include an external image file when publishing RDoc inside a Gem? RDoc keeps converting the image into html!

References

[1] http://www.ruby-doc.org/stdlib/libdoc/rdoc/rdoc/index.html
[2] http://www.graphviz.org/

Ruby Programming/Syntax

- Syntax
  - Lexicology
    - Identifiers
    - Comments
    - Embedded Documentation
    - Reserved Words
    - Expressions
  - Variables and Constants
    - Local Variables
    - Instance Variables
    - Class Variables
    - Global Variables
    - Constants
    - Pseudo Variables
    - Pre-defined Variables
  - Literals
    - Numerics
    - Strings
      - Interpolation
      - Backslash Notation
      - The % Notation
    - Command Expansion
    - Regular Expressions
    - Arrays
    - Hashes
    - Ranges
    - Symbols
  - Operators
    - Assignment
      - Self Assignment
      - Multiple Assignment
      - Conditional Assignment
    - Scope
    - and
    - or
    - not
  - Control Structures
    - Conditional Branches
      - if
        - if modifier
      - unless
        - unless modifier

        - case
    - Loops
        - while
            - while modifier
        - until
            - until modifier
        - for
            - for ... in
        - break
        - redo
        - next
        - retry
    - Exception Handling
        - raise
        - begin
            - rescue modifier
    - Miscellanea
        - return
- Method Calls
    - 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 Programming/Syntax/Lexicology

Identifiers

Example:

```
foobar
ruby_is_simple
```

An identifier in Ruby begins with an English letter (`A-Za-z`) or underscore (_) and consists of alphanumeric characters (`A-Za-z0-9`) or underscores (_). There is no restriction to its length. Identifiers are names used to identify variables, definitions, classes, etcetera in a program and from other other variables, definitions, …. An identifier cannot have a reserved word as its name .

Comments

Example:

```
# this is a comment line
```

Other than within a string, a comment is regarded as the text from the # to the end of the line.

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
```

Expressions

Example: true

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

All variables, literals, operators, 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 preceeding backslash (\) is continued to the following line.

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
```

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 substitute the value of a constant or trying to access an uninitialized constant raises the NameError exception.

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. (**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

```
$!          The exception information message set by 'raise'.
$@          Array of backtrace of the last exception thrown.
```

```
$&          The string matched by the last successful pattern match
in this scope.
$`          The string to the left  of the last successful match.
$'          The string to the right of the last successful match.
$+          The last bracket matched by the last successful match.
$1 to $9    The Nth group of the last successful regexp match.
$~          The information about the last match in the current scope.
```

```
$=          The flag for case insensitive, nil by default.
$/          The input record separator, newline by default.
$\          The output record separator for the print and IO#write.
Default is nil.
$,          The output field separator for the print and Array#join.
$;          The default separator for String#split.
```

```
$.          The current input line number of the last file that was
read.
$<          The virtual concatenation file of the files given on
command line.
$>          The default output for print, printf. $stdout by default.
$_          The last input line of string by gets or readline.
```

```
$0          Contains the name of the script being executed. May be
assignable.
$*          Command line arguments given for the script sans args.
$$          The process number of the Ruby running this script.
$?          The status of the last executed child process.
$:          Load path for scripts and binary modules by load or require.
```

```
$"          The array contains the module names loaded by require.
$DEBUG      The status of the -d switch.
$FILENAME   Current input file from $<. Same as $<.filename.
$LOAD_PATH  The alias to the $:.
$stderr     The current standard error output.
$stdin      The current standard input.
$stdout     The current standard output.
$VERBOSE    The verbose flag, which is set by the -v switch.
$-0         The alias to $/.
$-a         True if option -a ("autosplit" mode) is set. Read-only
variable.
$-d         The alias to $DEBUG.
$-F         The alias to $;.
$-i         If in-place-edit mode is set, this variable holds the
extension, otherwise nil.
$-I         The alias to $:.
```

```
$-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.
$-v        The alias to $VERBOSE.
$-w        True if option -w is set.
```

This infestation 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.

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 (\" for example)

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 (wacky) 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 choosen 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
```

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.

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.

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

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

Scope

Basically in Ruby there's a local scope, a global scope, and a class 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.

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

This works because prefixing a variable with a dollar sign makes the variable a global. The $ is ugly and the reason behind this is simply to make you not to use or use sparingly these global variables.

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

@instance_variable, instance of what? Of top level, which effectively makes it a global-like $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 class
 var.print_localvar             #=>undefined local variable or
method `localvar' for #<Test:0x2b36208 @instvar="kiwi">
(NameError).
```

This will print "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  instance of Test!!
 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  SubSubTest
 subtest.print_classvar
```

This new child of Test also has @@clasvar 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.

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.
```

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
```

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 if-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 if-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. 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) + "."
```

This [? (expr) : (expr)] is also known as the ternary operator. 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"
```

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 comparision:

```
 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"
puts 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
=>nil
```

Other ways to use case and variations on its syntax maybe seen at Linuxtopia Ruby Programming [1]

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

References

[1] http://www.linuxtopia.org/online_books/programming_books/ruby_tutorial/Ruby_Expressions_Case_Expressions.html

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.

```
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:

```
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 or the parameter's value is nil.

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

```
some_method('something')
```

The method call above will output:

```
something
0
```

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, high-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 a difficulty to understand 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.

Shivang's PROC

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:

```
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:

```
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:

```
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:

```
lamb = lambda {|x, y| puts x + y}
pnew = Proc.new {|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 way cooler a whopping two characters shorter, its behavior is less surprising.

Ruby's lambda is unusual in that choice of parameter names *does* affect behavior:

```
x = 3
lambda{|x| "x still refers to the outer variable"}.call(4)
puts x  # x is now 4, not 3
```

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:

```
# 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:

```
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:

```
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:

```
# 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):

```
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:

```
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:

```
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)
```

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 a 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, 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 or through the methods provided by that instance. They are accessed using the @ operator.

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.

Class Variables

Class variables are accessed using the @@ operator. These variables are associated with the class rather than any object instance of the class and are the same across all object instances. (These are the same as 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 [1]

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:

```
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:

```
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.

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

Declaring Visibility

By default, all methods in Ruby classes are public - accessible by anyone. If desired, this access 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.

As a result of that fact 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:

```
class Example
  def methodA
  end

  private # all methods that follow will be made private: not
accessible for outside objects

  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
explicit receiver at all, even if that receiver is "self"
    other_object.a # nope, a is private, you can't get it (but if it
was protected, you could!)
  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).

In Ruby, private visibility is what protected was in Java. Private methods in Ruby are accessible from children. This is a sensible design, since in Java, when method was private, it rendered it useless for children classes: making it a rule, that all methods should be "protected" by default, and never private. However, 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 [2] gives following example - implementing an operator that compares one internal variable with 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
automatically:
  # 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:

```
class ParentClass
  def a_method
    puts 'b'
  end
end

class SomeClass < ParentClass  # < means inherit (or "extends" if you are from Java bac
   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

puts X.new.foo
puts Y.new.foo
```

Outputs

```
hello
helloy
```

Mixing in Modules

First, you need to read up on modules. Modules are 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
```

OUTPUT : Modules ar included a1 is called b1 is called c1 is called

Notice the code.It 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.

References

[1] http://martinfowler.com/bliki/ClassInstanceVariable.html
[2] http://www.rubycentral.com/faq/rubyfaq-7.html

Ruby Programming/Reference

- Reference
  - Built-in Functions
  - Predefined Variables
  - Predefined Classes
    - Object
      - Array
      - Class
      - Exception
      - FalseClass
      - IO
        - File
          - File::Stat
      - Method
      - Module
        - Class
      - NilClass
      - Numeric
        - Integer
          - Bignum
          - Fixnum
        - Float
      - Range
      - Regexp
      - String
      - Struct
        - Struct::Tms
      - Symbol
      - Time
      - TrueClass
  - Standard Library

Ruby Programming/Reference/Predefined Variables

Ruby's predefined (built-in) variables affect the behavior of the entire program, so their use in libraries isn't recommended.
The values in most predefined variables can be accessed by alternative means.

`$!`

- The last exception object raised. The exception object can also be accessed using => in rescue clause.

`$@`

- The stack backtrace for the last exception raised. The stack backtrace information can retrieved by Exception#backtrace method of the last exception.

`$/`

- The input record separator (newline by default). gets, readline, etc., take their input record separator as optional argument.

`$\`

- The output record separator (nil by default).

`$,`

- The output separator between the arguments to print and Array#join (nil by default). You can specify separator explicitly to Array#join.

`$;`

- The default separator for split (nil by default). You can specify separator explicitly for String#split.

`$.`

- The number of the last line read from the current input file. Equivalent to ARGF.lineno.

`$<`

- Synonym for ARGF.

`$>`

- Synonym for $defout.

`$0`

- The name of the current Ruby program being executed.

`$$`

- The process.pid of the current Ruby program being executed.

`$?`

- The exit status of the last process terminated.

`$:`

- Synonym for $LOAD_PATH.

`$DEBUG`

- True if the -d or --debug command-line option is specified.

`$defout`

- The destination output for print and printf ($stdout by default).

`$F`

- The variable that receives the output from split when -a is specified. This variable is set if the -a command-line option is specified along with the -p or -n option.

`$FILENAME`

- The name of the file currently being read from ARGF. Equivalent to ARGF.filename.

`$LOAD_PATH`

- An array holding the directories to be searched when loading files with the load and require methods.

`$SAFE`

- The security level.

```
    0   No checks are performed on externally supplied (tainted) data.
(default)

    1   Potentially dangerous operations using tainted data are
forbidden.

    2   Potentially dangerous operations on processes and files are
forbidden.

    3   All newly created objects are considered tainted.

    4   Modification of global data is forbidden.
```

`$stdin`

- Standard input (STDIN by default).

`$stdout`

- Standard output (STDOUT by default).

`$stderr`

- Standard error (STDERR by default).

`$VERBOSE`

- True if the -v, -w, or --verbose command-line option is specified.

`$- x`

- The value of interpreter option -x (x=0, a, d, F, i, K, l, p, v).

The following are local variables:

`$_`

- The last string read by gets or readline in the current scope.

`$~`

- MatchData relating to the last match. Regex#match method returns the last match information.

The following variables hold values that change in accordance with the current value of $~ and can't receive assignment:

`$ n ($1, $2, $3...)`

- The string matched in the nth group of the last pattern match. Equivalent to m[n], where m is a MatchData object.

`$&`

- The string matched in the last pattern match. Equivalent to m[0], where m is a MatchData object.

$`

- The string preceding the match in the last pattern match. Equivalent to m.pre_match, where m is a MatchData object.

$'

- The string following the match in the last pattern match. Equivalent to m.post_match, where m is a MatchData object.

$+

- The string corresponding to the last successfully matched group in the last pattern match.

Ruby Programming/Reference/Predefined Classes

In ruby even the **base types** (also **predefined classes**) can be hacked.[1] In the following example, `5` is an immediate,[2] a literal, an object, and an instance of `Fixnum`.

```
class Fixnum

      alias :other_s :to_s
      def to_s()
            a = self + 5
            return a.other_s
      end

end


a = 5
puts a.class  ## prints Fixnum
puts a        ## prints 10 (adds 5 once)
puts 0        ## prints 5  (adds 5 once)
puts 5        ## prints 10 (adds 5 once)
puts 10 + 0   ## prints 15 (adds 5 once)

b = 5+5
puts b        ## puts 15 (adds 5 once)
```

Footnotes

1. ^ Which means the 4 VALUE bytes are not a reference but the value itself. All `5` have the same object id (which could also be achieved in other ways).
2. ^ Might not always work as you would like, the base types don't have a constructor (def initialize), and can't have simpleton methods. There are some other minor exceptions.

References

[1] http://en.wikipedia.org/wiki/Ruby_programming%2Freference%2Fpredefined_classes#endnote_quirks
[2] http://en.wikipedia.org/wiki/Ruby_programming%2Freference%2Fpredefined_classes#endnote_immediate

Ruby Programming/Reference/Objects

Object is the base class of all other classes created in Ruby. It provides the basic set of functions available to all classes, and each function can be explicitly overridden by the user.

This class provides a number of useful methods for all of the classes in Ruby.

Ruby Programming/Reference/Objects/Array

Class Methods

```
Method: [ ]        Signature: Array[ [anObject]* ] -> anArray
```

Creates a new array whose elements are given between [and].

```
 Array.[]( 1, 2, 3 ) -> [1,2,3]
 Array[ 1, 2, 3 ] -> [1,2,3]
 [1,2,3] -> [1,2,3]
```

Ruby Programming/Object/NilClass

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

Ruby Programming/Reference/Objects/Exception

Exception is the superclass for exceptions

Instance Methods

`backtrace`

- Returns the backtrace information (from where exception occurred) as an array of strings.

`exception`

- Returns a clone of the exception object. This method is used by raise method.

`message`

- Returns the exception message.

Ruby Programming/Reference/Objects/FalseClass

The only instance of FalseClass is false.
Methods:

```
false & other - Logical AND, without short circuit behavior
false | other - Logical OR, without short circuit behavior
false ^ other - Exclusive Or (XOR)
```

Ruby Programming/Reference/Objects/IO/File/File::Stat

First create a file, and then call the stat method on it. Use the methods of stat to obtain information about the file.

Example:

```
if File.new('/etc').stat.directory?
      puts '/etc is a directory'
end
```

Ruby Programming/Reference/Objects/Numeric

Numeric provides common behavior of numbers. Numeric is an abstract class, so it should not be instantiated.
Included Modules:

```
Comparable
```

Instance Methods:
+ n

```
    Returns n.
```

- n

```
    Returns n negated.
```

n + num
n - num
n * num
n / num

```
    Performs arithmetic operations: addition, subtraction,
multiplication, and division.
```

n % num

```
    Returns the modulus of n.
```

n ** num

```
    Exponentiation.
```

n.abs

```
    Returns the absolute value of n.
```

n.ceil

```
    Returns the smallest integer greater than or equal to n.
```

n.coerce(num)

```
   Returns an array containing num and n both possibly converted to a
type that allows them to be operated on mutually. Used in automatic
type conversion in numeric operators.
```

n.divmod(num)

```
    Returns an array containing the quotient and modulus from dividing
n by num.
```

n.floor

Returns the largest integer less than or equal to n.

```
    1.2.floor            #=> 1
    2.1.floor            #=> 2
    (-1.2).floor         #=> -2
    (-2.1).floor         #=> -3
```

n.integer?

```
    Returns true if n is an integer.
```

n.modulo(num)

```
    Returns the modulus obtained by dividing n by num and rounding the
quotient with floor. Equivalent to n.divmod(num)[1].
```

n.nonzero?

```
    Returns n if it isn't zero, otherwise nil.
```

n.remainder(num)

Returns the remainder obtained by dividing n by num and removing decimals from the quotient. The result and n always have same sign.

```
    (13.modulo(4))          #=>  1
    (13.modulo(-4))         #=> -3
    ((-13).modulo(4))       #=>  3
    ((-13).modulo(-4))      #=> -1
    (13.remainder(4))       #=>  1
    (13.remainder(-4))      #=>  1
    ((-13).remainder(4))    #=> -1
```

```
    (-13).remainder(-4))    #=> -1
```

n.round

```
Returns n rounded to the nearest integer.
```

```
    1.2.round               #=> 1
    2.5.round               #=> 3
    (-1.2).round            #=> -1
    (-2.5).round            #=> -3
```

n.truncate

```
Returns n as an integer with decimals removed.
```

```
    1.2.truncate            #=> 1
    2.1.truncate            #=> 2
    (-1.2).truncate         #=> -1
    (-2.1).truncate         #=> -2
```

n.zero?

```
    Returns zero if n is 0.
```

Ruby Programming/Reference/Objects/Numeric/Integer

Integer provides common behavior of integers (Fixnum and Bignum). Integer is an abstract class, so you should not instantiate this class.

Inherited Class: Numeric Included Module: Precision

Class Methods:

Integer::induced_from(numeric)

```
   Returns the result of converting numeric into an integer.
```

Instance Methods:

Bitwise operations: AND, OR, XOR, and inversion.

~i
i & int
i | int
i ^ int
i << int
i >> int

```
   Bitwise left shift and right shift.
```

`i[n]`

```
   Returns the value of the nth bit from the least significant bit,
which is i[0].
```

```
5[0]      # => 1
5[1]      # => 0
5[2]      # => 1
```

`i.chr`

    Returns a string containing the character for the character code i.

```
65.chr    # => "A"
?a.chr    # => "a"
```

`i.downto( min) {| i| ...}`

    Invokes the block, decrementing each time from i to min.

```
3.downto(1) {|i|
  puts i
}
```

```
# prints:
#  3
#  2
#  1
```

`i.next`
`i.succ`

    Returns the next integer following i. Equivalent to i + 1.

`i.size`

    Returns the number of bytes in the machine representation of i.

`i.step( upto, step) {| i| ...}`

    Iterates the block from i to upto, incrementing by step each time.

```
10.step(5, -2) {|i|
  puts i
}
# prints:
#  10
#  8
#  6
```

`i.succ`

    See i.next

`i.times {| i| ...}`

    Iterates the block i times.

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

```
# prints:
#  0
#  1
#  2
```

i.to_f

```
Converts i into a floating point number. Float conversion may lose
precision information.
```

```
1234567891234567.to_f   # => 1.234567891e+15
```

i.to_int

```
Returns i itself. Every object that has to_int method is treated as
if it's an integer.
```

i.upto(max) {| i| ...}

```
Invokes the block, incrementing each time from i to max.
```

```
1.upto(3) {|i|
  puts i
}
# prints:
#  1
#  2
#  3
```

Ruby Programming/Reference/Objects/Regexp

Class Regexp holds a regular expression, used to match a pattern of strings. Regular expressions can be created using /../ or by using the constructor "new".

CONSTANTS

1. EXTENDED : Ignore spaces and newlines in regexp.
2. IGNORECASE : Matches are case insensitive.
3. MULTILINE : Newlines treated as any other character.

CLASS METHODS

- compile : Regexp.compile(pattern [, options [lang]]) -> aRegexp

```
This method is a synonym for method "new".
```

- escape : Regexp.escape(aString) -> aNewString

```
Escapes any characters that have special meaning in a regular
expression.
   For example :
   Regexp.escape('\\*?{}.')  » \\\\\*\?\{\}\.
```

- last_match : Regexp.last_match -> aMatchData

```
Returns the MatchData object generated by the last successful pattern
match.
    Equivalent to reading the global variable $~.
```

- new : Regexp.new(pattern [, options [lang]]) -> aRegexp

```
Constructs a regular expression from the pattern. The pattern can be
either a string or
    a "regexp".
    For example:
    Regexp.new("xyz") » /xyz/
```

- quote : Regexp.quote(aString) -> aNewString

```
A synonym for escape method.
```

INSTANCE METHODS

- == : rxp == aRegexp -> true or false

```
Two regular expressions are equal if their patterns match and they have
 same character
    set code and theie casefold? values are same.
```

- === : rxp === aString -> true or false

```
Synonym for Regexp#=~ used in case statements
```

- =~ : rxp === aString -> true or false

```
Matches a regular expression with string and returns the offset of the
start of match or nil
    if a match fails.
```

- ~ : ~ rxp -> anInteger or nil

```
Match---Matches rxp against the contents of $_. Equivalent to rxp =~
$_.
    For example:
    $_ = "input data"
    ~ /at/  » 7
```

- casefold? : rxp.casefold? Returns true or false.

```
Returns the value of case-insensitive flag.
```

- kcode : rxp.kcode -> aString

```
Returns character set code for the regexp.
```

- match : rxp.match(aString) -> aMatchData or nil

```
Returns a MatchData object or nil if match is not found.
```

- source : rxp.source -> aString

```
Returns the original string of the pattern.
    For example : /xyz/.source >> xyz
```

Ruby Programming/Reference/Objects/String

String Class

Methods:

crypt(*salt*). Returns an encoded string using crypt(3). *salt* is a string with minimal length 2. If *salt* starts with "1", MD5 encryption is used, based on up to eight characters following "1". Otherwise, DES is used, based on the first two characters of *salt*.

Ruby Programming/Reference/Objects/Symbol

Symbols

A Ruby symbol is the internal representation of a name. You construct the symbol for a name by preceding the name with a colon. A particular name will always generate the same symbol, regardless of how that name is used within the program.

```
:Object
:myVariable
```

Other languages call this process ``interning, *and call symbols ``atoms.*

Ruby Programming/Reference/Objects/Time

```
class Tuesdays
      attr_accessor :time, :place

      def initialize(time, place)
            @time = time
            @place = place
      end
end

feb12 = Tuesdays.new("8:00", "Rice U.")
```

As for the object, it is clever let me give you some advice though. Firstly, you don't ever want to store a date or time as a string. This is always a mistake. -- though for learning purposes it works out, as in your example. In real life, simply not so. Lastly, you created a class called Tuesdays without specifying what would make it different from a class called Wednesday; that is to say the purpose is nebulous: there is nothing special about Tuesdays to a computer. If you have to use comments to differentiate Tuesdays from Wednesdays you typically fail.

```
class Event
      def initialize( place, time=Time.new )
            @place = place

            case time.class.to_s
                  when "Array"
```

```
                        @time = Time.gm( *time )
                  when "Time"
                        @time = time
                  else
                        throw "invalid time type"
            end
      end

      attr_accessor :time, :place

end

## Event at 5:00PM 2-2-2009  CST
funStart = Event.new( "evan-hodgson day",
[0,0,17,2,2,2009,2,nil,false,"CST"] )

## Event now, (see time=Time.new -- the default in constructor)
rightNow = Event.new( "NOW!" );

## You can compaire Event#time to any Time object!!
if Time.new > funStart.time
      puts "We're there"
else
      puts "Not yet"
end

## Because the constructor takes two forms of time, you can do
## Event.new( "Right now", Time.gm(stuff here) )
```

Ruby Programming/Reference/Objects/TrueClass

The only instance of TrueClass is true.
Methods:

```
true & other - Logical AND, without short circuit behavior
true | other - Logical OR, without short circuit behavior
true ^ other - Logical exclusive Or (XOR)
```

Ruby Programming/Reference/Built-in Modules

Enumerable

each_with_index

each_with_index calls its block with the item and its index.

```
array = ['Superman','Batman','The Hulk']

array.each_with_index do |item,index|
  puts "#{index} -> #{item}"
 end

# will print
# 0 -> Superman
# 1 -> Batman
# 2 -> The Hulk
```

find_all

find_all returns only those items for which the called block is not false

```
range = 1 .. 10

# find the even numbers

array = range.find_all { |item| item % 2 == 0 }

# returns [2,4,6,8,10]
```

```
array = ['Superman','Batman','Catwoman','Wonder Woman']

array = array.find_all { |item| item =~ /woman/ }

# returns ['Catwoman','Wonder Woman']
```

Ruby Programming/Built-in Modules

- Built-in Modules
  - Enumerable
  - Comparable
  - Precision
  - Math

Ruby Programming/GUI Toolkit Modules/Tk

Ruby bindings are available for several widget toolkits, among them Tk, Gtk, Fox, and Qt. The Tk binding is the oldest; it is widely available and still more or less the default toolkit for GUI programming with Ruby. Nevertheless, currently there exists no comprehensive manual for Ruby/Tk; the Ruby book recommends inferring Ruby/Tk usage from the Perl/Tk documentation.

The current Ruby "PickAxe book" has a chapter on Ruby/Tk.

Ruby Programming/GUI Toolkit Modules/ GTK2

- Ruby GTK API Reference [1]
- Ruby GTK Tutorial [2]

References

[1] http://ruby-gnome2.sourceforge.jp/hiki.cgi?Ruby%2FGTK
[2] http://zetcode.com/tutorials/rubygtktutorial/

Ruby Programming/GUI Toolkit Modules/Qt4

[Totally free download and tutorial on Ruby Qt4 bindings [1]]

[Prag Prog guys' pdf book on Qt3 for Ruby [2]]

References

[1] http://www.darshancomputing.com/qt4-qtruby-tutorial/

[2] http://www.pragmaticprogrammer.com/titles/ctrubyqt/

Ruby Programming/XML Processing/REXML

REXML is a XML processing API. As of Ruby 1.8, it is included in the Standard API.

REXML can read and write XML documents. Validation against a DTD or a schema is not yet fully implemented.

Basics

DOM

Definitions

Using the DOM API, REXML can parse documents and build a tree containing the **element**s, **attribute**s, and **text**s.

For example, this might be used to save a wikibook:

```
<wikibook title="Programming:Ruby">

</wikibook>
```

In this case, the chapter is an **element**. It has an **attribute** title with the **value** Overview and a **text** with the **value** "Ruby is a programming language [...]".

The section is also an element. It has an attribute, too, but no text. Instead, it has an element, the chapter element.

In short, elements can have attributes, text and child elements.

Representation in REXML

When parsing a XML document, an instance of the REXML::Document class is created. (The new message of REXML::Document just has to be fed with a REXML::Document itself, or a String, or an IO.) This represents the whole document, including the <?xml...?> tag. REXML::Document itself is a subclass of **REXML::Element**, an important class.

When using DOM, instances of the Element class are representing the elements of the XML document. They might have attributes, accessible using the attributes message, text, and child elements.

The Document is an Element itself, but usually, you might be more interested in the root element of the XML document. As defined in the XML specification, any document has only one root element; it can be easily obtained calling REXML::Document.root().

Once you have obtained the root Element, you can go down the tree using the elements message defined in Element, which returns a collection of all child elements, or access attributes or texts, whatever you need.

The tree can be modified, too. In addition, the to_s methods have been overridden to return the XML code of elements, attributes and text. Element.to_s returns the XML code of the whole element, including attributes, text, and

child elements' XML code. You can call that on the Document, too.

External links

Standard API Documentation at ruby-doc.org, including the rexml package [1]

References

[1] http://www.ruby-doc.org/stdlib/

Ruby on Rails/Print version

Note: the current version of this book can be found at http://en.wikibooks.org/wiki/Ruby_on_Rails"

Introduction

Ruby on Rails, or often seen as RoR or just Rails, is an web application framework written in the programming language Ruby. Rails was created to be a project management tool for "37signals [1]". Quickly, it became one of the most popular frameworks on the web and its ideas and philosophies has inspired many frameworks in different languages.

Features

- Rails supports agile software development [2]
- Rails has a short and very intuitive language
- Single blocks of code are intended to be short und clear
- Rails is very flexible
- Rails comes with a testing framework
- The framework follows the principles of DRY [3] (Don't repeat yourself)
- Rails has some conventions: these allow quick development and consistent code
- Rails uses the MVC-Architecture [4] (Model-View-Controller)
- Rails comes with an integrated server for local testing
- Applications with Rails are RESTful [5]

Getting Started

Installation on Windows

To start, you will need the following components:

- Ruby
- RubyGems
- Rails
- a driver for your database

Ruby

Install Ruby as a regular application. You might need administrator privileges to install it successfully. Check the Ruby site for the latest version. The Ruby sites provides packages for all OS. Just follow their steps: ruby website [1]

Gems

RubyGems is a packaged Ruby application. Using the `gem` command helps you installing and deleting gem-packages. Gems allows you to install additional features for your application and easy management of already installed ones.

Download the latest RubyGem package from the offical ruby website [6]. To install, open up a console and navigate to the folder containing the gem file and the file `setup.rb`. To install the gems, type:

```
ruby setup.rb
```

To verify the installation, check the version of gems:

```
gem -v
```

It should display the proper version (1.3.1 as of the writing of this book)

Rails

To install Rails, we can use our freshly installed gems (Rails is a gem). Use the console to type

```
gem install rails
```

This downloads and install all the needed components on your system. After the installation is done, verify the installation by checking if you have the latest version:

```
rails -v
```

It should display the current Rails version (2.3.2 as of writing this guide)

DB Driver

Rails supports a broad range of databases. The default database is SQLite3. You can specify the database you want to use when you first create your Rails project. But no worries, it can be altered any time. For this Wikibook, we will use SQLite3 as database.

To use SQLite3 on your system together with Ruby on Rails, download the latest dll files from the sqlite website [7]. Choose the files that contains the dlls without TCL binding sqlitedll-3 x x.zip. After downloading, copy all the files inside the zip into your Ruby `/bin` directory

At last, we need to install a gem for our database. Since we are using SQLite3 in this book, we want to install the proper gem:

```
gem install sqlite3-ruby --version 1.2.3
```

Even though 1.2.3 is not the current version, it will work as intended because newer version won't work in Windows

Database Configuration

If your first application is created, take a look inside the `/config` folder. We need to tell Rails the name of our database and how to use it. Depending on your chosen database during the creation of the Rails application, the `database.yml` file always look different. Because we decided to stay with the default SQLite3 database, the file will look something like this:

```
# SQLite version 3.x
# gem install sqlite3-ruby (not necessary on OS X Leopard)
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000

# Warning: The database defined as <tt>test</tt> will be erased and
# re-generated from your development database when you run <tt>rake</tt>.
# Do not set this db to the same as development or production.
test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000

production:
  adapter: sqlite3
  database: db/production.sqlite3
  pool: 5
  timeout: 5000
```

We may now give our database a proper name. Consider giving different names to your "test"- and to your "production"-database. This is especially important when working in a live-environment and you do not want bad data inside your database.

To create your database in our new project environment, we need to run

```
rake db:create
```

Rake is a build in tool, that allows you to run many pre-made programs to ease up development. There is more functionality inside Rake then just creating a database. But one thing at a time. The `db:create` commando creates the database with the given name (inside db.yml). If you're using SQLite3, you will find a *.sqlite3 file in the `/db` folder. This is your database, stored in a single, handy file. For practicing and local development purposes, this is ideal because all your data is in one file that can be quickly read and written.

Install on OS X

Option 1

- Download Ruby and RubyGems [8]
- Install Ruby
- Install RubyGems
- Get to the command-line and type:

```
sudo gem install rails --include-dependencies
```

Option 2

- Download Locomotive [9]
- Install Ruby on Rails software from Locomotive

Option 3

- Use the broken Ruby installation on OSX 10.4 and follow this guide [10]

RMagic

A shellscript has been made that will install everything you need to make RMagic work on your OS X box. Copy the contents, save it as a file and run the file through a terminal (by navigating to the file and type `./name_of_file`.

Install on Linux

Install or update Ruby

It is important that you get the right version of the language. Versions 1.8.2 and 1.8.4 are recommended. 1.8.3 has problems, and 1.8.5 (the current stable version) runs fine, but you will not be able to use breakpointer for debugging (a very handy tool). So I recommend 1.8.4.

Get it from the offical Ruby website [1]

Get it from the SVN repository [11]

Debian

For Debian-based systems (Debian, Ubuntu, etc.), your best bet is to type:

```
sudo apt-get install ruby -t '1.8.4'
sudo apt-get install irb rdoc
```

This should install the Ruby language interpreter, the RDoc documentation generator, and irb command-line interpreter.

Install RubyGems

You should be able to obtain RubyGems by going to the Gems Website [12] and clicking the "download" link. Choose the proper archive and install.

Install Rails

Get to the command-line and type:

```
sudo gem install rails --include-dependencies
```

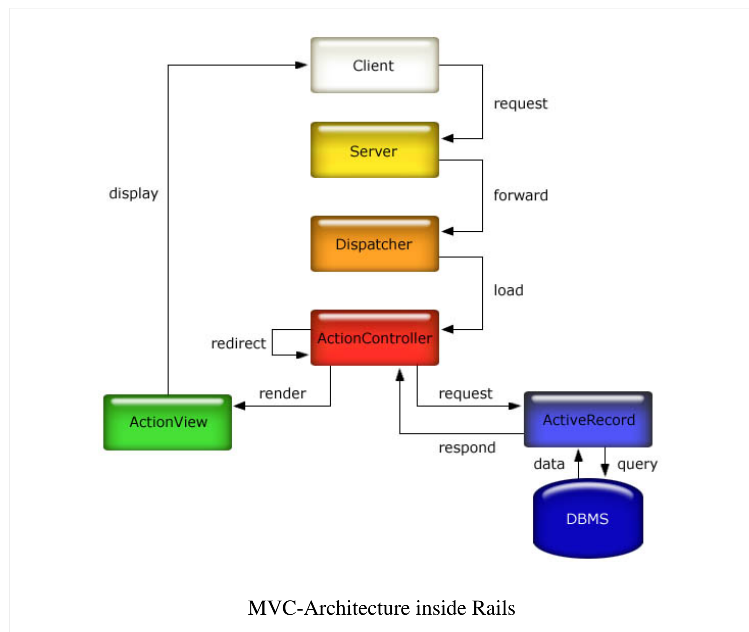
Don´t repeat yourself

To help to maintain clean code, Rails follows the idea of DRY. The idea behind it, is simple: whenever possible, re-use as much code as possible. This reduces errors, keeps your code clean and even more important: it takes a lot of work off your shoulders by writing code once and using it again. For more information of DRY look at the Wikipedia article [3]

Model-View-Controller

As already mentioned, Rails relies on the MVC pattern. Model-View-Controller has some benefits over traditional concepts:

- it keeps your business logic separated from your (HTML-based) views
- keeps your code clean and neat in one place



MVC-Architecture inside Rails

Model

The model represents the information and the data from the database. It is as independent from the database as possible (Rails comes with its own O/R-Mapper, allowing you to change the database that feeds the application but not the application itself). The model also does the validation of the data before it gets into the database. Most of the time you will find a table in the database and an according model in your application.

View

The view is the presentation layer for your application. The view layer is responsible for rendering your models into one or more formats, such as XHTML, XML, or even Javascript. Rails supports arbitrary text rendering and thus all text formats, but also includes explicit support for Javascript and XML. Inside the view you will find (most of the time) HTML with embedded Ruby code. In Rails, views are implemented using *ERb* by default.

Controller

The controller connects the model with the view. In Rails, controllers are implemented as *ActionController* classes. The controller knows how to process the data that comes from the model and how to pass it onto the view. The controller should not include any database related actions (such as modifying data before it gets saved inside the database). This should be handled in the proper model.

Helpers

When you have code that you use frequently in your views or that is too big/messy to put inside of a view, you can define a method for it inside of a helper. All methods defined in the helpers are automatically usable in the views.

Best Practices

Current best practices include:

- fat model and skinny controller
- business logic should always be in the model
- the view should have minimal code
- Use helpers!

Convention over Configuration

When starting to work with Rails you will find yourself looking at controllers and lots of views and models for your database. In order to reduce the need of heavy configuration, the team behind Rails has set up rules to ease up working with the application. These rules are not one-way. You may define your own rules but for the beginning (and for your further work, too) it's a good idea to stick to the conventions that Rails offers. These conventions will speed up development, keep your code concise and readable and - most important - these conventions allow you an easy navigation inside your application.

An example should show you how the conventions work together: You have a database table called `orders` with the primary key `id`. The matching model is called `order` and the controller, that handles all the logic is named `orders_controller`. The view is split in different actions: if the controller has a `new` and `edit` action, there is also a new- and edit-view.

First Application

Create a basic Rails Application

Creating your first Rails application is as simple as typing:

```
rails firstapplication
```

in the console. This will create the Rails application in the current directory. If you need to use another database than the default (SQLite3) you can specify the database with the `-d` parameter. If you need e.g MySQL support, create your app with

```
 rails firstapplication -d mysql
```

NOTE: on Windows, you have to to start the commands with `ruby` explicitly. On Linux and OSX you can just use the command without starting `ruby`.

This sets up the basic files and folders you need. Let's have a look at the files and folders and what they are good for:

The Application Structure

After creating your Rails application, you will be presented with a directory structure. The following table should introduce you to the folders and their purpose:

File/Folder	What is it good for
app/	This is the folder where you will work most of the time. It contains your model, view and the controller.
app/model	Contains all your models for your project.
app/view	Includes all HTML files inside a folder. These files are named after an corresponding action inside your controller. Aditionally, these files are grouped into folders with the controller name.
app/controller	Holds the logic for your Rails application.
db/	All database related files go into this folder. If you are working with SQLite, then the database is likely to be in this folder as *.sqlite3 file.
db/migrate	Has all the migrations that you created.
config/	As the names suggest, it has all the necessary configuration files of your Rails application. There is only very little configuration needed, so no worries that you have to crawl through endless lines of code. Localizations, routes and all basic files can be found here.
public/	All your static files (files that do not change dynamically) your CSS or JavaScript files are inside this folder.
public/images	
public/javascripts	When you embedd Javascript, CSS or images, the default location is inside these folders. Rails will automatically look inside these to find the proper file.
public/stylesheets	
script/	Holds scripts for Rails that provide a variety of tasks. These scripts link to another file where the "real" files are. Inside these folder you will find the generators, server and console.
log/	Log files from your application. If you want to debug some parts of your application you can check out these files.
test/	All files for testing your application.
doc/	Documentation for the current Rails application.
lib/	Extended modules for your application.
vendor/	Whenever you have 3rd-party plug ins, they will be found in this folder.
tmp/	Temporary files
README	Basic guide for others on how to setup your application, what specialities it has or what to be careful about.
Rakefile	Handles all the Rake tasks inside your application.

Basic creation of different files

Most of Rails files can be created via the console by entering a simple command that tells Rails what you want. This way you can create database migrations, controllers, views and much more. All commands look like

```
ruby script/generate generator generator-options
```

To see what options are available, just enter

```
ruby script/generate
```

and Rails will show you all available options and what generators are currently installed. By default, you can choose from the different generators. The most important are:

- controller
- helper

- mailer
- migration
- model
- scaffold

If you want more information on different generators, simply enter the generator command e.g. `script/generate model` in the console and you will get information for this specific command and examples explaining what the generator does.

When working with generators, Rails will create and name all the necessary files in a proper way according to the conventions. This is especially important when working with Migrations (see later). Rails will never overwrite your files if they already exist (unless you tell Rails explicitly to do so).

A very good way to get you started with everything you need is to `scaffold` your files. This will create a basic CRUD-structure for your files. (CRUD=**C**reate **R**ead **U**pdate and **D**elete; this reflects SQL attributes create, insert, update and delete) Scaffolding creates not only your migrations but also a controller with the most basic syntax and a view-folder, that comes with templates for very basic displaying and editing of your data. To use scaffolding, we will use our generators. The scaffolding generator expects the name of a `MODEL/CONTROLLER` and the proper fields and types (in the format field:type).

Say, we want to create tags for different products, we can use:

```
ruby script/generate scaffold Tag name:string popularity:integer
```

Inside the console, Rails will give us information about the files it created:

```
     exists  app/models/
     exists  app/controllers/
     exists  app/helpers/
     create  app/views/tags
     exists  app/views/layouts/
     exists  test/functional/
     exists  test/unit/
     exists  test/unit/helpers/
     exists  public/stylesheets/
     create  app/views/tags/index.html.erb
     create  app/views/tags/show.html.erb
     create  app/views/tags/new.html.erb
     create  app/views/tags/edit.html.erb
     create  app/views/layouts/tags.html.erb
     identical  public/stylesheets/scaffold.css
     create  app/controllers/tags_controller.rb
     create  test/functional/tags_controller_test.rb
     create  app/helpers/tags_helper.rb
     create  test/unit/helpers/tags_helper_test.rb
     route  map.resources :tags
     dependency  model
     exists    app/models/
     exists    test/unit/
     exists    test/fixtures/
     create    app/models/tag.rb
     create    test/unit/tag_test.rb
```

```
     create    test/fixtures/tags.yml
     exists    db/migrate
     create    db/migrate/20090420180053_create_tags.rb
```

Let's take a quick tour of the files we have now: First Rails created a separate folder inside the view named `tags`. Inside this folder we have some different templatefiles. Besides, we have a controller (`tags_controller.rb`), files for tests (`tags_controller_test.rb, tags_helper_test.rb, tags.yml, tag_test.rb`), helpers (`tags_helper.rb`), a migration file (`20090420180053_create_tags.rb`) and last but not least, a route was also added. As you can see, scaffold does quite a lot of work for us. But remember, that the generated code is very basic and far from being "good" but it already gives you a good starting point.

Running the server

The bundled Webrick server

As you already know, Rails comes with an integrated server: WEBrick. WEBrick is a Ruby-written server to gets you started right from the beginning. There are alternatives such as Mongrel or Phusion Passenger (formerly known as mod_ruby, a module for Apache). For local(!!) development WEBrick is a good choice.

To start up the server, simply open up a console, navigate to your Rails application and type

- On Windows:

```
 ruby script/server
```

- On OS X and Linux:

```
$ script/server
```

After some seconds, WEBrick has been initialized and you are ready to go. The console with the web server needs to stay open, otherwise the server will shut down. To see if everything is working as expected, open up your web browser and switch to

```
http://localhost:3000
```

You should see the default Rails start page saying that everything is working correct. You can view the details page (name) for the current version of your environment and some other variables. The server console not only runs the server, but shows how the requests by the browser are processed, including the amount of queries, the used SQL syntax or the data from your submitted forms.

There are several options, including but not limited to:

- `-p port`: Specify the port to run on
- `-b ip`: Bind to a specific IP address
- `-e name`: Use a specific Rails environment (like `production`)
- `-d`: Run in daemon mode
- `-h`: Display a help message with all command-line options

Mongrel

To start a single mongrel instance:

- On all platforms:

```
mongrel_rails start
```

This should be executed in the root directory of the Rails app you wish to run on Mongrel. There are numerous options you can specify, including:

- `-p port`: run on a specific port
- `-e environment`: execute with a specific Rails environment, like `production`
- `-d`: run in daemon mode

Built-In Rails Tools

Generators

Until Make a generator is created, here the link to the next page: Ruby on Rails/Built-In Rails Tools/What is Rake anyway?

Generators

Introduction

Rails comes with a number of generators which are used to create stub files for models, controllers, views, unit tests, migrations and more. Generators are accessed through the command-line script `RAILS_ROOT/script/generate`

All commands look like

```
 ruby script/generate generator generator-options
```

To see what options are available, just enter

```
 ruby script/generate
```

and Rails will show you all available options and what generators are currently installed. By default, you can choose from the different generators. The most important are:

- controller
- helper
- mailer
- migration
- model
- scaffold

If you want more information on different generators, simply enter the generator commend e.g. "`script/generate model`" in the console and you will get information to this specific command and examples explaining what the generator does.

You can use the generator multiple times for the same controller, but be careful: it will give you the option to overwrite your controller file (to add the actions you specify). As long as you have not modified the controller this might be fine, but if you have already added code then make sure you do not overwrite it and go back and manually add the action methods.

Generate a Model

To generate a model use the following:

```
ruby script/generate model ModelName column:datatype column:datatype
[...]
```

Replace ModelName with the CamelCase version of your model name. For example:

```
ruby script/generate model People name:string age:integer
```

This would generate the following:

```
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/people.rb
create  test/unit/people_test.rb
create  test/fixtures/peoples.yml
exists  db/migrate
create  db/migrate/20090607101912_create_peoples.rb
```

Any necessary directories will be automatically created. Existing ones will not be replaced. The file `app/models/people.rb` contains the People class. `test/unit/people_test.rb` contains the unit test stub for the People class. `test/fixtures/peoples.yml` contains a stub for test data which will be used to populate the test database during test runs. The `db/migrate/20090607101912_create_peoples.rb` contains the database migration stub for the Person class. Note the the timestamp at the beginning of the file (`20090607101912`) will always be different depending on the time you created the file.

Generate a Controller

To generate a controller use the following:

```
ruby script/generate controller ControllerName [actions]
```

Replace ControllerName with the CamelCase version of your controller name. When no actions are given, Rails will create a Controller that responds to all 7 REST actions (new, create, update, edit, destroy, index & show)

```
ruby script/generate controller People
```

would generate the following output:

```
exists  app/controllers/
exists  app/helpers/
create  app/views/people
exists  test/functional/
exists  test/unit/helpers/
create  app/controllers/people_controller.rb
create  test/functional/people_controller_test.rb
create  app/helpers/people_helper.rb
create  test/unit/helpers/people_helper_test.rb
```

The file `app/controllers/people_controller.rb` contains the PeopleController class. `test/functional/people_controller_test.rb` contains the functional test stub for the PersonController class. `app/helpers/people_helper.rb` is a stub for helper methods which will be made

available to that controller and its associated views. Inside `app/views/people` you will find the created templates for the controller. Depending on the given parameters, there will be different files.

Generate a Migration

To generate a migration use the following:

```
 ruby script/generate migration MigrationName column:datatype
column:datatype [...]
```

Replace MigrationName with the CamelCase version of your migration name. For example:

```
 script/generate migration AddCityToPerson
```

This would generate the following:

```
exists  db/migrate
create  db/migrate/20090607103358_add_city_to_person.rb
```

Migration are automatically generated each time you construct a new model, so you do not need to generate migrations by hand for each model. Typically you will use the migration generator when you need to change an existing model or need join tables. Again, the timestamp starting the filename will be different.

Rake

Rake [13] is a Ruby build tool like make and Ant.

- Rakefiles (rake's version of Makefiles) are completely defined in standard Ruby syntax. No XML files to edit. No quirky Makefile syntax to worry about (is that a tab or a space?)
- Users can specify tasks with prerequisites.
- Rake supports rule patterns to synthesize implicit tasks.
- Flexible FileLists that act like arrays but know about manipulating file names and paths.
- A library of prepackaged tasks to make building rakefiles easier.

You can find out what tasks are currently available to Rake with the `rake -T` command. The tasks below are common and used on regular basis.

Database Tasks

- `rake db:migrate [VERSION=x]`: Execute the migrations to the specified version. If the version is omitted then it will migrate to the highest version possible.
- `rake db:sessions:clear`: Clear all of the sessions in the database.

Test Tasks

- `rake test:units`: Execute unit tests

Cache Tasks

- `rake tmp:cache:clear`: Clear out the page/fragment cache.

You can make your own rake task by creating a file in the `lib/tasks` directory of your Rails application and adding the Rake tasks to it. For example, adding the following to `lib/tasks/database.rake` will make the `db:recreate` task available to your Rails application:

```
  namespace :db do
    desc "Drop and create the current database"
    task :recreate => :environment do
      abcs = ActiveRecord::Base.configurations
      ActiveRecord::Base.establish_connection(abcs[RAILS_ENV])

ActiveRecord::Base.connection.recreate_database(ActiveRecord::Base.connection.current_dat
    end
  end
```

The namespace method puts the contents of the block in the specified namespace. You can nest namespaces as deep as you want although usually one or two nestings is sufficient.

This task can now be executed with

```
rake db:recreate
```

Rake can be executed with the following command-line options:

- `--dry-run` or `-n`: Do a dry run without executing actions
- `--help` or `-H`: Display a help message
- `--libdir=LIBDIR` or `-I LIBDIR`: Include LIBDIR in the search path for required modules
- `--rakelibdir=RAKELIBDIR` or `-R RAKELIBDIR`: Auto-import any .rake files in RAKELIBDIR. (default is 'rakelib')
- `--nosearch` or `-N`: Do not search parent directories for the Rakefile
- `--prereqs` or `-P`: Display the tasks and dependencies, then exit
- `--quiet` or `-q`: Do not log messages to standard output
- `--rakefile=FILE` or `-f FILE`: Use FILE as the rakefile
- `--require=MODULE` or `-r MODULE`: Require MODULE before executing rakefile
- `--silent` or `-s`: Like `--quiet`, but also suppresses the 'in directory' announcement
- `--tasks[=PATTERN]` or `-T [PATTERN]`: Display the tasks (matching optional PATTERN) with descriptions, then exit
- `--trace` or `-t`: Turn on invoke/execute tracing, enable full backtrace
- `--usage` or `-h`: Display usage
- `--verbose` or `-v`: Log message to standard output (default)
- `--version` or `-V`: Display the program version
- `--classic-namespace` or `-C`: Put Task and FileTask in the top level namespace

ActiveRecord - The Model

Naming

Basics

ActiveRecord uses convention for naming classes, tables and fields. Rails uses convention over configuration. ActiveRecord expects applications to follow certain naming conventions. These conventions extend from file naming, class naming, table naming and more. By default classes are singular, tables are plural, primary keys are `id` and foreign keys are *`table`*`_id`.

Note: There are also certain names that are reserved and should not be used in your model for attributes outside of the conventions defined in Rails:

- `lock_version`
- `type` - This is only used when you have single table inheritance and must contain a class name
- `id` - Reserved for primary keys
- `table_name_count` - Reserved for counter cache
- `position` - Reserved for acts_as_list
- `parent_id` - Reserved for acts_as_tree
- `lft` - Reserved for acts_as_nested_set
- `rgt` - Reserved for acts_as_nested_set
- `quote` - Method in ActiveRecord::Base which is used to quote SQL
- `template`

Classes

ActiveRecord classes are named in singular form.

Tables

Tables for ActiveRecord objects are named in plural form by default. This pluralization is often an initial point of contention for new Rails users.

If you need to change the table name there are several ways to override the default behavior.

Set use_pluralization

In `config/environment.rb` you can specify `ActiveRecord::Base.use_pluralization = false`. This will apply to all ActiveRecord objects.

Use set_table_name

You can call `set_table_name` to specify a custom table name for a particular model.

For example:

```
class Dog < ActiveRecord::Base
  set_table_name 'dog'
end
```

Override table_name

You can also override the `table_name` method and return a value.

For example:

```
class Dog < ActiveRecord::Base
  def table_name
    'dog'
  end
end
```

Migrations

[14] Migrations meant to solve the problem of rolling out changes to your database. By defining the changes to your database schema in Ruby files, development teams can ensure that all changes to the database are properly versioned. Additionally migrations help to ensure that rolling out changes to fellow developers as well as other servers (development, QA, production) is handled in a consistent and manageable fashion.

Building a Migration

You can either build the migration on its own using

```
ruby script/generate migration Category
```

and write the specific commands afterwards (if you want to create custom SQL, this is the way to go) or you can create a model that comes with the migration using

```
ruby script/generate model Category name:string amount:integer
```

The console tells you that there were some files created and some already existent. As mentioned before, Rails will never overwrite existing files unless stated otherwise.

Now lets take a look at the migration

```
# 20090409120944_create_categories.rb
class CreateCategories < ActiveRecord::Migration
  def self.up
    create_table :categories do |t|
      t.string :name
      t.integer :amount

      t.timestamps
    end
  end

  def self.down
    drop_table :categories
  end
end
```

First of all, take a look at the number (*20090409120944*) in front of your file. This is the timestamp of your file and important for the creation of the database tables. This timestamp will always be different, depending on the exact time of the creation of your migration. The idea behind this is to have a "history" of all your migrations available.

But why is this important?

Imagine that you work on a Rails project and you create tables, alter columns or remove columns from your database via migrations. After some time, your client changes his mind and he wants only very basic features and you already started to create advanced features and altered the database. Because you can't remember all the changes that went into the database and their order, you will either spent a lot of time working on the database to have the "old" state available or you have to start from scratch because it would take too long to remember and redo all changes. This is where migration come in handy, because of the timestamp, Rails is able to recognize the changes in their actual order and all changes can be undone easily. Never alter the timestamp manually. This will certainly cause problems. For more on those topic, check out the "managing migrations" section

Speaking of undoing and redoing: notice the two methods inside your migration `self.up` and `self.down`. Both of them do exactly the opposite of each other. While `self.up` creates our categories table with all columns, `self.down` removes (drops) the table from the database with all its contents(!!). When Rails sees that the migration has not been moved to the database, it will use the `self.up` method, if you undo the migration, the `self.down` method gets executed. This way you can make sure that you will always be able to go back to a past state of your database. Keep in mind when writing own migrations always include a `self.up` and a `self.down` method to assure that the database state will be consistent after an rollback.

OK, let's start with the migration itself:

```
create_table :categories do |t|
      t.string :name
      t.integer :amount
      t.timestamps
end
```

We want to create a table called categories(`create_table :categories`) that has a name and an amount column. Additionally Rails adds an timestamp for us where it will store the creation date and the update date for each row. Rails will also create an primary key called **model**_id that auto-increments (1,2,3,...) with every row.

You can choose from a variety of datatypes that go with ActiveRecord. The most common types are:

- string
- text
- integer
- decimal
- timestamp
- references
- boolean

But wait, there is not yet a single table nor column in our database. We need to write the migration file into the database.

```
rake db:migrate
```

handles this job. The command is able to create the table and all the necessary columns inside the table. This command is not limited to migrating a single file so you can migrate an unlimited number of files at once. Rake also knows what migrations are already in the database so it won't overwrite your tables. For more info see "Managing Migrations".

To add a connection between your tables we want to add references in our model. References are comparable to foreign keys (Rails doesn't use foreign keys by default because not all databases can handle foreign keys but you can write custom SQL to make use of foreign keys) and tell your table where to look for further data.

Let's add another model to our already existent database. We want to create a category that has multiply products. So we need to reference this product in our category. We want to create a model:

```
ruby script/generate model Products name:string category:references
```

and insert it into the database

```
rake db:migrate
```

Note the type :references for the category. This tells Rails to create a column inside the database that holds a reference to our category. Inside our database there is now a `category_id` column for our product. (In order to work with these two models, we need to add associations inside our models, see Associations)

Managing Migrations

We already talked about how migrations can help you to organise your database in a very convenient manner. Now we will take a look at how this is achieved. You already know that the timestamp in the filename tells rails when the migration was created and Rake know what migrations are already inside the database.

To restore the state of the databse as it was, say 5 migrations before the current, we can use

```
$rake db:rollback STEP=5
```

This will undo the last 5 migrations that have been comitted to the database.

To redo the last 5 steps, we can use a similar command

```
$ rake db:migrate:redo STEP=5
```

You can also rollback or redo a specific version of a migration state, you just need to povide the timestamp:

```
$ rake db:migrate:up VERSION=20080906120000
```

Choose wether you want the db_migrate:up method to be executed or the db_migrate:down method

Keep in mind, that restoring your database to a previous state will delete already inserted data completely!

Schema Method Reference

The following methods are available to define your schema changes in Ruby:

- `create_table(name, options)`: Creates a table called name and makes the table object available to a block that adds columns to it, following the same format as `add_column`. See example above. The options hash is for fragments like "DEFAULT CHARSET=UTF-8" that are appended to the create table definition.
- `drop_table(name)`: Drops the table called name.
- `rename_table(old_name, new_name)`: Renames the table called old_name to new_name.
- `add_column(table_name, column_name, type, options)`: Adds a new column to the table called table_name named column_name specified to be one of the following types: :string, :text, :integer, :float, :datetime, :timestamp, :time, :date, :binary, :boolean. A default value can be specified by passing an options hash like { :default => 11 }.
- `rename_column(table_name, column_name, new_column_name)`: Renames a column but keeps the type and content.
- `change_column(table_name, column_name, type, options)`: Changes the column to a different type using the same parameters as `add_column`.
- `remove_column(table_name, column_name)`: Removes the column named column_name from the table called table_name.

- `add_index(table_name, column_names, index_type, index_name)`: Add a new index with the name of the column, or index_name (if specified) on the column(s). Specify an optional index_type (e.g. UNIQUE).
- `remove_index(table_name, index_name)`: Remove the index specified by index_name.

Command Reference

- `rake db:create[:all]`: If :all not specified then create the database defined in config/database.yml for the current RAILS_ENV. If :all is specified then create all of the databases defined in config/database.yml.
- `rake db:fixtures:load`: Load fixtures into the current environment's database. Load specific fixtures using FIXTURES=x,y
- `rake db:migrate [VERSION=n]`: Migrate the database through scripts in db/migrate. Target specific version with VERSION=n
- `rake db:migrate:redo [STEP=n]`: (2.0.2) Revert the database by rolling back "STEP" number of VERSIONS and re-applying migrations.
- `rake db:migrate:reset`: (2.0.2) Drop the database, create it and then re-apply all migrations. The considerations outlined in the note to rake db:create apply.
- `rake db:reset`: Drop and re-create database using db/schema.rb. The considerations outlined in the note to rake db:create apply.
- `rake db:rollback [STEP=N]`: (2.0.2) Revert migration 1 or n STEPs back.
- `rake db:schema:dump`: Create a db/schema.rb file that can be portably used against any DB supported by AR
- `rake db:schema:load`: Load a schema.rb file into the database
- `rake db:sessions:clear`: Clear the sessions table
- `rake db:sessions:create`: Creates a sessions table for use with CGI::Session::ActiveRecordStore
- `rake db:structure:dump`: Dump the database structure to a SQL file
- `rake db:test:clone`: Recreate the test database from the current environment's database schema
- `rake db:test:clone_structure`: Recreate the test databases from the development structure
- `rake db:test:prepare`: Prepare the test database and load the schema
- `rake db:test:purge`: Empty the test database

You can obtain the list of commands at any time using `rake -T` from within your application's root directory.

You can get a fuller description of each task by using

```
rake --describe task:name:and:options
```

See also

The official API for Migrations [15]

Associations

Introduction

Associations are methods to connect 2 models. Operations on objects become quite simple. The association describes the role of relations that models are having with each other. ActiveRecord associations can be used to describe one-to-one (1:1), one-to-many (1:n) and many-to-many (n:m) relationships between models. There are several types of associations

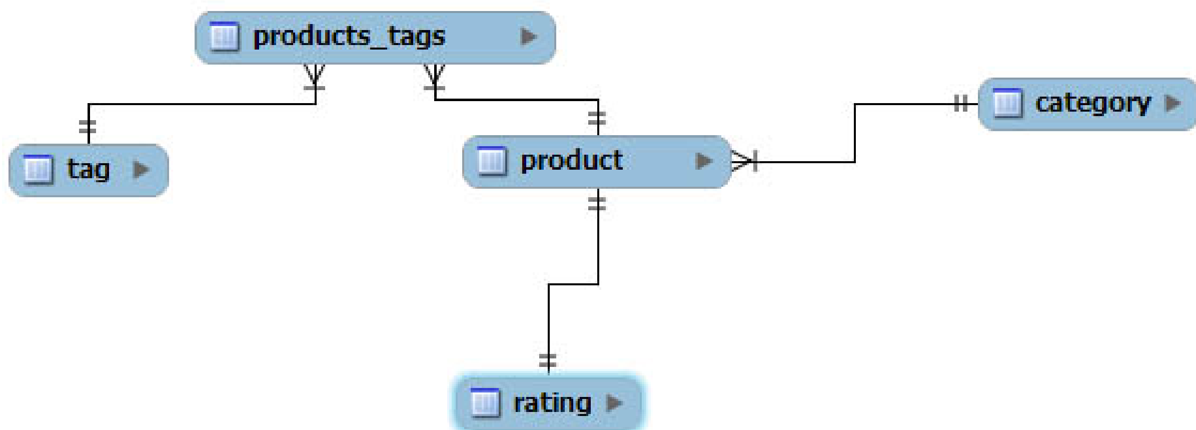
- *belongs_to* and
- *has_one* form a one-to-one relationship

- *has_one :through* is a different way to create a one-to-one relationship
- *has_many* and
- *belongs_to* form a one-to-many relation
- *has_and_belongs_to_many* or an alternative way
- *has_many :through* to create a many-to-many relationship

For all following examples, we assume the following 4 models:

```
class Product< ActiveRecord::Base
end

class Category < ActiveRecord::Base
end

class Rating< ActiveRecord::Base

end

class Tag< ActiveRecord::Base
end
```

Also check out the ER-diagramm for the models below:



belongs_to

The `belongs_to` association sets up an one-to-one (1:1) or one-to-many (1:n) relationship to an other model.

In our example, one Rating belongs to exactly one Product, so we need to set up a `belongs_to` association.

```
class Product < ActiveRecord::Base
  belongs_to: rating
end
```

The product stores an id for a rating, both of them have a one-to-one Relationship. Meaning: One product belongs to one category and one product belongs to one rating.

has_one

The `has_one` assocation is a one-to-one relation too. It declares that each instance of a product from the productmodel possesses only one rating_id.

```
class Rating< ActiveRecord::Base
  has_one:product
end
```

Since the rating and the product form a one-to-one relationship, it is up to us where we put the reference. Because we have put the reference (the rating_id) into the product (see belongs_to :rating into the product model), the association for the rating has to be has_one:product. `Has_one` and `belongs_to` always go together to form a relationship.

has_many

This is a one-to-many connection to an other model. In this case, one product has many categories. Pay attention to the spelling, if you use `has_many`. The model that you want to reference needs to be written in plural.

```
class Category< ActiveRecord::Base
  has_many:products #note the plural here!
end

class Product< ActiveRecord::Base
  belongs_to:category
end
```

The category and the product are forming a one-to-many relationship. Since a category has many products we put the `has_many` association inside the category. **Note** that also the product needs to contain a reference to the category. One product belongs to exactly one category, therefore we put `belongs_to :category` inside the product model.

has_and_belongs_to_many

Also known as : *HABTM*.

`Has_and_belongs_to_many` is the most complex association. You need to keep some things in mind: Because of the way relational databases work, you can not set up a direct relationship. You need to create a joining table. This is easily done with migrations. If we want to create a HABTM association for our product and tag, the association for the joining table might look like following:

```
class CreateProductTagJoinTable < ActiveRecord::Migration
  def self.up
   create_table :products_tags, :id => false do |t| #we DO NOT need
the id here!
     t.integer :product_id #alternatively, we can write t.references
:product
     t.integer :tag_id
   end
  end

  def self.down
    drop_table :products_tags
  end
```

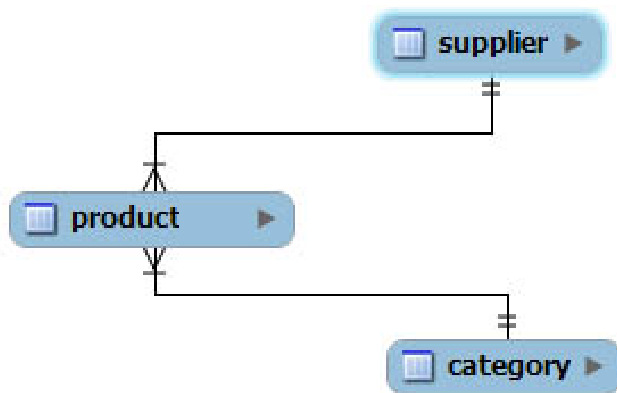
```
end
```

Because we do not need a primary key inside the join table, we use `:id => false` to prevent the automatic creation of a primary key column. The naming of the table needs to be in alphabetical order. "*P*" is before "*T*" in the alphabet so the tablename has to be products_tags (note: plural).

```
class Product< ActiveRecord::Base
  has_and_belongs_to_many :tags
end

class Tag< ActiveRecord::Base
   has_and_belongs_to_many :products
end
```

has_many :through

This association is an other way to form a many-to-many relation. Consider the following ER-Diagramm to show the relationship.



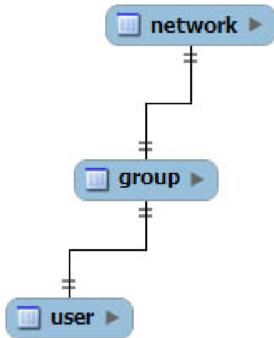

```
class Product < ActiveRecord::Base
  belongs_to :category
  belongs_to :supplier
end

class Supplier< ActiveRecord::Base
  has_many :products
  has_many :categories, :through => :products
end

class Category< ActiveRecord::Base
  has_many :products
  has_many :suppliers, :through => :products
end
```

Instead of using a join table we use another model for the relationship. If we want to find out which supplier is responsible for a specific category we need the product to link the models together. This way we can store additional data inside a real model that also acts as a joining model. Whenever you need to store data that additionally belongs to association (e.g. the date of the creation) you might want to use this type of association.

has_one :through

As with `has_many :through`, the `has_one :through` association uses an "extra" model to form a relation.

Consider this visual relationship:




```
class Network< ActiveRecord::Base
  has_one :group
  has_one :user, :through =>:group
end

class Group< ActiveRecord::Base
  belongs_to :network
  has_one :user
end

class User< ActiveRecord::Base
  belongs_to :group
end
```

Note: This example assumes that the membership inside a group is exlusive for each user.

Callbacks

Callbacks provide a means of hooking into an ActiveRecord object's lifecycle.

Implementing Callbacks

There are four types of callbacks accepted by the callback macros:

- Method references (symbol)
- Callback objects
- Inline methods (using a proc)
- Inline eval methods (using a string) - deprecated.

Method references and callback objects are the recommended approaches, inline methods using a proc are sometimes appropriate (such as for creating mix-ins) and inline eval methods are deprecated.

Method Reference

The method reference callbacks work by specifying a protected or private method available in the object, like this:

```
class Topic < ActiveRecord::Base
  before_destroy :delete_parents

  private
    def delete_parents
      self.class.delete_all "parent_id = #{id}"
    end
end
```

Callback Objects

The callback objects have methods named after the callback, called with the record as the only parameter such as:

```
class BankAccount < ActiveRecord::Base
  before_save      EncryptionWrapper.new("credit_card_number")
  after_save       EncryptionWrapper.new("credit_card_number")
  after_initialize EncryptionWrapper.new("credit_card_number")
end

class EncryptionWrapper
  def initialize(attribute)
    @attribute = attribute
  end

  def before_save(record)
    record.credit_card_number = encrypt(record.credit_card_number)
  end

  def after_save(record)
    record.credit_card_number = decrypt(record.credit_card_number)
  end

  alias_method :after_find, :after_save

  private
    def encrypt(value)
      # Secrecy is committed
    end

    def decrypt(value)
      # Secrecy is unveiled
    end
end
```

So you specify the object you want messaged on a given callback. When that callback is triggered the object has a method by the name of the callback messaged.

Proc

Example of using a Proc for a callback:

```
class Person
  before_save Proc.new { |model| model.do_something }
end
```

Inline Eval

The callback macros usually accept a symbol for the method they're supposed to run, but you can also pass a "method string" which will then be evaluated within the binding of the callback. Example:

```
class Topic < ActiveRecord::Base
  before_destroy 'self.class.delete_all "parent_id = #{id}"'
end
```

Notice that single plings (') are used so the #{id} part isn't evaluated until the callback is triggered. Also note that these inline callbacks can be stacked just like the regular ones:

```
class Topic < ActiveRecord::Base
  before_destroy 'self.class.delete_all "parent_id = #{id}"',
                 'puts "Evaluated after parents are destroyed"'
end
```

Callback Reference

before_save

This method is called before an ActiveRecord object is saved.

before_create

called before creating a new object of the model

Partially Documented Callbacks

The following callbacks are partially documented. Their use is discouraged because of [16] performance issues.

after_find

The after_find callback is only executed if there is an explicit implementation in the model class. It will be called for each object returned from a find, and thus can potentially affect performance as noted in the [16] Rails API Documentation.

after_initialize

The after_initialize method is only executed if there is an explicit implementation in the model class. It will be called whenever an ActiveRecord model is initialized. It will be called for each object returned from a find, and thus can potentially affect performance as noted in the [16] Rails API documentation.

References

[1] http://www.37signals.com
[2] http://en.wikipedia.org/wiki/Agile_development
[3] http://en.wikipedia.org/wiki/Don%27t_repeat_yourself
[4] http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
[5] http://en.wikipedia.org/wiki/Representational_State_Transfer
[6] http://rubyforge.org/frs/?group_id=126
[7] http://www.sqlite.org/download.html
[8] http://www.rubyonrails.org/down
[9] http://locomotive.sourceforge.net/
[10] http://hivelogic.com/narrative/articles/ruby-rails-mongrel-mysql-osx
[11] http://svn.ruby-lang.org/repos/ruby/tags/1_8_4/
[12] http://docs.rubygems.org/
[13] http://rake.rubyforge.org/
[14] **Excerpts modified and republished from Steve Eichert's Ruby on rails Migrations Explained** (http://www.emxsoftware.com/RubyOnRails/Ruby+on+Rails+Migrations+Explained) **article.**
[15] http://api.rubyonrails.org/classes/ActiveRecord/Migration.html
[16] "Because after_find and after_initialize are called for each object found and instantiated by a finder, such as Base.find(:all), we’ve had to implement a simple performance constraint (50% more speed on a simple test case). Unlike all the other callbacks, after_find and after_initialize will only be run if an explicit implementation is defined (def after_find). In that case, all of the callback types will be called."

=

Validations

ActiveRecord supports a variety of model validation methods and allows for addition new methods as needed.

General Usage

For a complete reference of all validations check out the official guide (http://guides.rubyonrails.org/activerecord_validations_callbacks.html#validation-helpers) or the API documentation (http://api.rubyonrails.org/classes/ActiveRecord/Validations/ClassMethods.html)

Additionally you can apply a validation to one or more attributes using the `validate_each` directive:

```
class Person < ActiveRecord::Base
  validates_each :first_name, :last_name do |record, attr, value|
    record.errors.add attr, 'starts with z.' if value[0] == ?z
  end
end
```

It is also possible to define validations via custom methods or blocks:

- validate
- validate_on_create
- validate_on_update

For example:

```
  class Person < ActiveRecord::Base
    validate :validate_email
```

```
      def validate_email
        record.errors.add :email, 'Invalid email' unless email =~ /@/
      end
    end
```

Normally validation occurs whenever a record is created or saved, however you can force validation to *not* occur using the `save_with_validation` method passing @false@ as the argument.

Important Validators

validates_acceptance_of

```
validates_acceptance_of :play_rules
```

If you need to check if the user has set or "accepted" a certain checkbox you can use this validation. In this case, the check box with the HTML attribute "name='play_rules'" needs to be checked in order to pass validation.

validates_confirmation_of

This validator checks if an input field has been entered correct both times. For example if you want the user to enter his password 2 times to make sure he enters the correct password (this is seen very often when registering for a site) this is the helper you want to use. To use it, make sure to define it correctly in your view (Note the **_confirmation**):

```
<%= text_field :user, :password%>
<%= text_field :user, :password_confirmation %>
```

validates_format_of

`validates_format_of` accepts a regular expression and checks for the input with the provided pattern given by the :with clause. Also note, that we used a customized message with this helper. This can be done with every validator in the same manner.

```
validates_format_of :username, :with => /\A[a-zA-Z]+\z/,  :message
=> "Please use only regular letters as username"
```

validates_length_of/validates_size_of

```
validates_length_of :username, :minimum => 5, :maximum => 50
validates_size_of :address, :in => 5..100
```

The `length_of/size_of` validator comes in handy if you need to check the input for a certain length of characters. In the example above, the username should not be longer than 50 characters and not shorter than 5. Alternatively you can specify a range that should be true in order to vaildate. Above the address should constist of 5 to 100 characters. Note that `length_of` and `size_of/` are the same.

validates_numericality_of

```
validates_numericality_of :amount_available
```

Checks if the input is a number of any kind. To make sure, that the input is only an integer, you may use the optional `:only_integer => true` command. There are some useful options with this command for example `:greater_than => 100` makes sure that the given number will be greater then 100: so 101 will be the first valid number.

validates_presence_of

```
validates_presence_of :name
```

One of the most basic validators, this one checks if anything has been inserted in the given form element ("name" in this case).

validates_uniqueness_of

```
validates_uniqueness_of: username
```

Finally, a bit of a more advanced validator: this one checks inside the database if the attribute is already taken or not. In this case, if the user chooses the username "Paul" and the name "Paul" already exists inside the username column of your database, it won't validate.

with scope

It can also validate whether the value of the specified attributes are unique based on multiple scope parameters. For example, making sure that a teacher can only be on the schedule once per semester for a particular class.

```
validates_uniqueness_of :teacher_id, :scope => [:semester_id,
:class_id]
```

When the record is created a check is performed to make sure that no record exists in the database with the given value for the specified attribute (that maps to a column). When the record is updated, the same check is made but disregarding the record itself.

Configuration Options

- message - Specifies a custom error message (default is: "has already been taken")
- scope - One or more columns by which to limit the scope of the uniqueness constraint.
- if - Specifies a method, proc or string to call to determine if the validation should occur (e.g. :if => :allow_validation, or :if => Proc.new { |user| user.signup_step > 2 }). The method, proc or string should return or evaluate to a true or false value.

When writing your validation keep in mind that you can mix all of these together and there are even more advanded functions (http:/ / guides. rubyonrails. org/ activerecord_validations_callbacks. html#conditional-validation) you might want to check out.

Attributes

ActiveRecord attributes are automatically determined from the underlying database schema.

Basic Usage

All of the columns in a table are accessible through methods on the ActiveRecord model. For example:

Consider the following migration:

```
# 20090409120944_create_products.rb
class CreateProducts < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
      t.string :name
      t.float :price
      t.integer :amount
```

```
    end
  end
#...
end
```

Because Rails has created a primary key for us and called it "id" we can already search for it inside the database. We can interact with the database via the build-in console:

```
 ruby script/console
```

gives us a console that let us communicate with Rails.

Before we can select data from our database it is time to insert some products

```
>> Product.new(:name => "my book", :price => 14.99, :amount
 => 4).save
```

if everything went right, Rails will show you a `=>true` inside the console... Let's add some more products

```
 >> Product.new(:name => "my movie", :price => 4.89 :amount
 => 1).save
```

```
 >> Product.new(:name => "my cd", :price => 9.98, :amount
=> 12).save
```

Now we have 3 products in our database, going from 1 to 3...

So let's select the first entry

```
 >>Product.find(1)
```

and Rails tells you what it found for this id (remember: find only works when there is a Rails generated id as primary key)

```
 => #<Product: id=1, name="my book" ..... >
```

so we are able to tell Rails to look for our entry with a specific id. But what happens when we have thousands of entries in our database? We can't remember every id of a Product, so Rails offers a very nifty solution:

```
 >>Product.find_by_name("my book")
```

```
 => #<Product: id=1, name="my book" ..... >
```

and Rails gives as the output we wanted. This not only works for the name, but for all columns in the database. We could also use `.find_by_price` or `.find_by_amount`

Or you can search for multiple products with their id:

>>Product.find(1,3)

=> #<Product: id=1, name="my book" ..... ><Product: id=3, name="my cd" ..... >

You can also search for all products

```
 >>Product.all
```

or

```
 >>Products.find(:all)
```

for the first or the last product in your table

```
>>Product.first/last
```

or

```
>>Product.find(:first/:last)
```

If you want more controll over your data, you can use many build-in options or use custom SQL. Let's try to find our data in descending order (3,2,1)

```
>>Product.all(:order => "id DESC")
```

or alternatively

```
>>Product.find(:all, :order => "id DESC")
```

will give you all your prodcuts, starting with the last inserted id - in our case: 3

When we query our database we get lots of infos we are not really interested in such as the "created_at" and "updated_at" column. We only want to take a look at our id and the name so we may use

```
>>Product.all(:select => "id, name")
```

or

```
>>Product.find(:all, :select => "id, name")
```

This method will only display the name and the id of our products making it way more readabel.

When searching for data, keep in mind that you can combine all these methods to your liking. To see more possibilities check out the RoR guide (http://guides.rubyonrails.org/active_record_querying.html#find-options) or the API (http://api.rubyonrails.org/classes/ActiveRecord/Base.html#M002208).

Overriding Attributes

You can override any of the accessor methods in your ActiveRecord model classes in order to add logic. For example:

```
class Product
  def name
    self[:name] || 'Unknown'
  end
end
```

```
p = Product.new
```

```
p.name
```

```
=> "Unknown"
```

```
p.name = 'my disk'
```

```
p.name
```

```
=> "my disk"
```

You can access any attribute via `self[attribute_name]` (to get) or `self[attribute_name]=` (to set).

Aggregations

Active Record implements aggregation through a macro-like class method called `composed_of` for representing attributes as value objects. It expresses relationships like "Account [is] composed of Money [among other things]" or "Person [is] composed of [an] address". Each call to the macro adds a description of how the value objects are created from the attributes of the entity object (when the entity is initialized either as a new object or from finding an existing object) and how it can be turned back into attributes (when the entity is saved to the database).

Example:

```
  class Customer < ActiveRecord::Base
    composed_of :balance, :class_name => "Money", :mapping =>
%w(balance amount)
    composed_of :address, :mapping => [ %w(address_street street),
%w(address_city city) ]
  end
```

The customer class now has the following methods to manipulate the value objects:

```
  * Customer#balance, Customer#balance=(money)
  * Customer#address, Customer#address=(address)
```

These methods will operate with value objects like the ones described below:

```
 class Money
   include Comparable
   attr_reader :amount, :currency
   EXCHANGE_RATES = { "USD_TO_DKK" => 6 }

   def initialize(amount, currency = "USD")
     @amount, @currency = amount, currency
   end

   def exchange_to(other_currency)
     exchanged_amount = (amount *
EXCHANGE_RATES["#{currency}_TO_#{other_currency}"]).floor
     Money.new(exchanged_amount, other_currency)
   end

   def ==(other_money)
     amount == other_money.amount && currency ==
other_money.currency
   end

   def <=>(other_money)
     if currency == other_money.currency
       amount <=> amount
     else
       amount <=> other_money.exchange_to(currency).amount
     end
   end
```

```
end

class Address
  attr_reader :street, :city
  def initialize(street, city)
    @street, @city = street, city
  end

  def close_to?(other_address)
    city == other_address.city
  end

  def ==(other_address)
    city == other_address.city && street ==
other_address.street
  end
end
```

Now it's possible to access attributes from the database through the value objects instead. If you choose to name the composition the same as the attributes name, it will be the only way to access that attribute. That's the case with our balance attribute. You interact with the value objects just like you would any other attribute, though:

```
customer.balance = Money.new(20)     # sets the Money value object and
the attribute
customer.balance                     # => Money value object
customer.balance.exchanged_to("DKK") # => Money.new(120, "DKK")
customer.balance > Money.new(10)     # => true
customer.balance == Money.new(20)    # => true
customer.balance < Money.new(5)      # => false
```

Value objects can also be composed of multiple attributes such as the case of Address. The order of the mappings will determine the order of the parameters. Example:

```
customer.address_street = "Hyancintvej"
customer.address_city   = "Copenhagen"
customer.address        # => Address.new("Hyancintvej",
"Copenhagen")
customer.address = Address.new("May Street", "Chicago")
customer.address_street # => "May Street"
customer.address_city   # => "Chicago"
```

Writing value objects

Value objects are immutable and interchangeable objects that represent a given value such as a Money object representing $5. Two Money objects both representing $5 should be equal (through methods such as == and <=> from Comparable if ranking makes sense). This is unlike entity objects where equality is determined by identity. An entity class such as Customer can easily have two different objects that both have an address on Hyancintvej. Entity identity is determined by object or relational unique identifiers (such as primary keys). Normal `ActiveRecord::Base` classes are entity objects.

It’s also important to treat the value objects as immutable. Don‘t allow the Money object to have its amount changed after creation. Create a new money object with the new value instead. This is exemplified by the Money#exchanged_to method that returns a new value object instead of changing its own values. Active Record won’t persist value objects that have been changed through other means than the writer method.

The immutable requirement is enforced by Active Record by freezing any object assigned as a value object. Attempting to change it afterwards will result in a TypeError.

Calculations

Calculations provide methods for calculating aggregate values of columns in ActiveRecord models.

Calculate

All calculations are handled through the calculate method. The calculate method accepts the name of the operation, the column name and any options. The options can be used to customize the query with :conditions, :order, :group, :having and :joins.

The supported calculations are:

- average
- sum
- minimum
- maximum
- count

The calculate method has two modes of working. If the :group option is *not* set then the result will be returned as a single numeric value (fixnum for count, float for average and the column type for everything else). If the :group option is set then the result is returned as an ordered Hash of the values and groups them by the :group column. The :group option takes either a column name or the name of a `belongs_to` association.

Note that if a condition specified in the calculation results in no values returned from the underlying table then the calculate method will return `nil`.

For example:

```
values = Person.maximum(:age, :group => 'last_name')
puts values["Drake"]
=> 43
```

```
drake  = Family.find_by_last_name('Drake')
values = Person.maximum(:age, :group => :family) # Person
belongs_to :family
puts values[drake]
=> 43
```

```
values.each do |family, max_age|
   ...
end
```

Average

You can use the average method to calculate the average value for a particular column. For example:

```
Person.average(:age)
```

Will return the average age of all people in the Person model.

An example of customizing the query:

```
Person.average(:age, :conditions => ['age >= ?', 55])
```

This would return the average age of people who are 55 or older.

Sum

The sum method will calculate the sum for a particular column. For example:

```
Product.sum(:number_in_stock)
```

Will return the sum of products in stock.

An example of customizing the query:

```
Product.sum(:number_in_stock, :conditions => ['category_id =>
?', 10])
```

Will return the sum of the number of products which are in category 10 and in stock.

Minimum

The minimum method will calculate the minimum value for a particular column. For example:

```
Donation.minimum(:amount)
```

Will return the lowest amount donated.

An example of customizing the query:

```
Donation.minimum(:amount, :conditions => ['created_at > ?',
1.year.ago])
```

This will return the lowest amount donated within the last year.

Maximum

The maximum method will calculate the maximum value for a particular column. For example:

```
Donation.maximum(:amount)
```

Will return the largest amount donated.

An example of customizing the query:

```
Donation.maximum(:amount, :conditions => ['created_at > ?',
1.year.ago])
```

This will return the largest amount donated within the last year.

Count

Counts the number of items matching the condition(s).

```
TestResult.count(:all)
```

Will return the number of TestResult objects in the database.

An example of customizing the query:

```
TestResult.count(:all,:conditions=>['starttime>=?',Time.now-3600*24])
```

Will return the number of TestResult objects whose starttime field is within the last 24 hours.

ActionView - The View

Rendering and Redirecting

Introduction

You already know how to manage your data with ActiveRecord. Now it is time to display your data. All data the view displays comes from the controller. Most of the time, you will work with HTML but you can also use Javascript inside your views (which of course can again be Rails generated) or different CSS.

The "Convention over Configuration" is also an essential part of the view: as mentioned in the beginning of this book, Rails is able to know which file for the view belongs to which action inside in the controller:

```
app/controller/products_controller.rb
```

```
#...
def show
  @product= Product.find(params[:id])
end
#...
```

This action inside our products controller assumes that there is a view responding to the name:

```
app/views/products/show.html.erb
```

As you can see the file has 2 extensions: one is for the browser to display (HTML) and the other one tell Rails how to process it (erb= embedded ruby).

Rendering and Redirecting

You will come across 2 often used methods to display data `render` and `redirect_to` A good example of how these two methods work can be seen in the example below:

This actions gets called when the user submitted updated data

```
def update
    @product= Product.find(params[:id])

    if @product.update_attributes(params[:name])
      redirect_to :action => 'index'
    else
      render :edit
```

```
    end
end
```

As you can see both of our methods are used in this simple example. Whenever we successfully updated the name of a product, we get redirected to the index site of the products. If the update fails, we want to return to the edit view.

There is an important difference between `render` and `redirect_to`: render will tell Rails what view it should use (with the same parameters you may have already sent) but `redirect_to` sends a new request to the browser.

Render

Remember the "update" action above? When the update fails, we want to render the edit view with the exact same parameters as we did before, in this case we look for the "id" inside the database and populate the page accordingly. If you want to render another view use

```
render 'categories/show'
```

You can also display a file that is somewhere completely different on your web server

```
render :file => "/u/apps/some_folder/app/views/offers/index"
```

And of course, you can render simple text

```
render :text => "Hello World"
```

You may have already noticed that there is a "layout" folder inside your view. Whenever you use scaffolding to create parts of your application a file inside layout gets created. If you scaffold "Products" the file inside layout will be called `products.html.erb`. This file is responsible for the basic display of your sites matching the common name (in this example, it's products). Whenever you want to redirect your user to another layout you can use

```
render :layout => 'another_layout'
```

Whenever there is no proper layout file, Rails will display the page only with the styling provided inside the requested view. To use a specific layout inside your whole controller you can define the layout inside your Controller

```
class ProductsController < ApplicationController
  layout "my_layout"
  #our actions
end
```

For more infos about layouts, see "Layout Files"

redirect_to

You can use `redirect_to` in a similar manner as you do with render, but keep the big difference between `render` and `redirect_to` in mind.

With `redirect_to` you can easily send the user to a new resource, for example the index page of our products. To learn more about the paths and routing see the chapter on "routing"

```
 redirect_to products_path
```

A very handy `redirect_to` option is `:back`

```
 redirect_to :back
```

will send the user back to the site that he came from

Templates

There are several templating systems included with Rails each designed to solve a different problem.

- ERb - Embedded Ruby is the default templating system for Rails apps. All files ending with `.rhtml` are considered ERb templates.
- Builder - Builder templates are programmatic templates which are useful for rendering markup such as XML. All templates ending with `.rxml` are treated as builder templates and include a variable called `xml` which is an instance of XmlMarkup.

In addition to the built in template systems you can also register new template handlers using the `ActionView::Base.register_template_handler(extension, class)` method. A template handler must implement the initialize(base) method which takes the `ActionView::Base` instance and a `render(text, locals)` method which takes the text to be rendered and the hash of local variables. **=**

Layout Files

In the previous chapter you learned how to render output to the default layout or to a special layout provided by you. You may have already looked inside such a layout file. Its default content will be similar to

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8"
/>
  <title>Products: <%= controller.action_name
%></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>
<p style="color: green"><%= flash[:notice] %></p>
<%= yield %>
</body>
</html>
```

So you now notice some things that are typical for a layout: the head section has an embedded tag for a stylesheet and the complete content is represented using a yield-tag

Asset tags

All asset tags, such as the "`stylesheet_link_tag`" are little helpers that generate the proper HTML for you. You can use this tags to embed CSS, Java Script and images or to provide RSS-feeds.

To embedded Javascript or CSS, you can use the according helpers:

```
<%= javascript_include_tag "my_javascript" %>
```

or if you want to include CSS

```
<%= stylesheet_link_tag "my_css" %>
```

All these files are located in the public directory `public/javascripts` or `public/stylesheets`. There is no need to provide the extension as Rails will handle it automatically. If you want to include multiple files just

provide all the file names inside the tags:

```
<%= stylesheet_link_tag "my_css", "my_other_css" %>
```

Of course these files can be placed anywhere in your application, just be sure to provide the proper path:

```
<%= stylesheet_link_tag "my_css", "files/stylesheet" %>
```

To load all files inside your `public/javascripts` or `public/styleheets` use the appropriate tag with `:all`

```
<%= javascript_include_tag :all %>
```

When embedding CSS you can specify the media attribute inside the tag:

```
<%= stylesheet_link_tag "my_print", media => "print" %>
```

No modern web site is complete without using pictures and graphices so Rails provides you with its own image tag:

```
<%= image_tag "my_image" %>
```

As with JavaScript or CSS the Rails will look into `public/images` by default to find the proper picture. **Note:** that you do not need to tell Rails if you use .jpg, .gif or .png! You can also provide a custom path and HTML attributes

```
<%= image_tag "images/my_image", :height => 50, :width => 50,
:alt => "This is my image" %>
```

Yield

Whenever you come across "yield" you will know that this is sort of a placeholder for the view that will be displayed inside your layout, but you are not limited to using a single yield:

```
<body>
  <%= yield :header %>
</body>
```

To get access to the `:header` you can use the "`content_for`" method. So with the above example, that might be:

```
<% content_for :header do %>
  <h1>Hello Rails - This is my header</h1>
<% end %>
```

Named yields can be used for menus, footers or sidebars to ease up maintainance. Just put them in your code and they will be placed in the right place.

Partials

It is time for a little more DRY. If you have a lot of HTML code that repeats itself in many different views (such as forms, dynamic menus,...) you might want to consider using partials: With partials, you just move the code that you use often into a separate file and place a link inside the files that should use your partials

```
<%= render :partial => "form" %>
```

This will look inside the folder from where the partial gets called for a file named `_form.html.erb`. Keep in mind to use an underscore in your filename to tell Rails that it is a partial.

To move your partial to a seperate folder use

```
<%= render :partial => "partials/form" %>
```

will look in "partials" for a file called `_form.html.erb`

Overall partials are a very good way to minimize the effort of copying the same code over and over, but to keep the needed code in single files that can be altered easily and used as often as you want to.

Forms

Basic

Forms are an important part of every web site. We use them to contact the site owners to provide log in credentials or to submit data. Rails has vast support for forms and many build in tags to help you create forms quick and easily.

The most basic form you can get is a simple input field for your name:

```
<% form_tag do %>
  <%= label_tag(:my_name, "My name is:") %>
  <%= text_field_tag(:my_name) %>
  <%= submit_tag("Process") %>
<% end %>
```

This will create a very basic form like

```
<form action="/products/test/1" method="post">
 <div style="margin:0;padding:0">
 <input name="authenticity_token" type="hidden"
value="KBBEVzSZOkCi/s9LKhYvW/gdwyZzwH2n39py5FggaT4=" />
 </div>

 <label for="my_name">My name is:</label>
 <input id="my_name" name="my_name" type="text" />
 <input name="commit" type="submit" value="Process" />
</form>
```

This was achieved by creating a simple action called `test.html.erb` inside our products view,

```
def test
  #nothing here yet
end
```

An empty action inside the controller was added. The page was called inside the browser http://localhost:3000/products/test/1. Because we haven't altered the routes we have to provide an id of a product. Additionaly Rails has

created a "authenticity_token" to provide security features to our form. You also notice that there is an id for our input. Rails provides the id for every form element with the names you provided. Of course this can be altered by providing the proper option.

The following list should give you an overview of some form tags. As most of the tags work in a similar manner, it should be easy to figure out how much of them works. Also be sure to take a look at the official API (http://api.rubyonrails.org/classes/ActionView/Helpers/FormTagHelper.html#M001706).

Radio Boxes:

To create radioboxes simply use

```
<%= radio_button_tag :category, "books", true %>
```

The above will create

```
<input id="category_books" name="category" type="radio"
value="books" />
```

If you want to change the id you can pass your own id to to tag `(:id => "someid")`. To preselect an item the 3 option that the helper receives is supposed to be a boolean.

Submit Button:

```
<%= submit_tag ("Save products", :class => "button") %>
```

will create a submit button

```
<input name="commit" type="submit" class="button" value="Save
products" />
```

Text fields:

```
<%= text_field_tag ('rails') %>
```

will create an emtpy text field:

```
<input id="rails" name="rails" type="text" />
```

As usual you can add HTML options or specific values

```
<%= text_field_tag ('price', nil, :maxlength => 10) %>
```

notice that we use "`nil`" here. This is because we want to create a text field with no pre-defined value:

```
<input id="price" name="price" type="text" maxlength="10" />
```

If we would have provided a name, Rails would add the HTML attribute "value":

```
<input id="price" name="price" type="text" maxlength="10" value="my
predefined value instead of nil" />
```

Select boxes:

A common task inside forms is to let the user select something from select boxes - or drop downs. To display a simple select box we can combine some handy methods:

```
<%= select_tag (:name,
options_for_select([['Peter','pete'],['Joseph', 'jo']]))%>
```

would create

```
<select id="name" name="name">
 <option value="pete">Peter</option>
 <option value="jo">Joseph</option>
</select>
```

Another nice helper of the `select_*` family is `select_year`. When you want the user to choose his year of birth, you may use something like:

```
<%= select_year Date.today, :start_year => 2009, :end_year =>
1900  %>
```

This will create a select box with all years, starting from 2009 going to 1900 with the current year (`Date.today` as the first argument) being pre-selected. Be sure to check out the other helpers for dates Datehelpers (http://api.rubyonrails.org/classes/ActionView/Helpers/DateHelper.html)

Handling Errors

Now that you know how to display forms correctly in Rails, we are looking for a way to give the user some feedback over his input. Either if it is correct or if he needs to re-enter some values. You have already read about how to validate form data before it gets written into the database. Now you will see how to display the errors. Basically there are two methods that allow us to display errors: `error_messages` and `error_messages_for`. The name gives a pretty good idea of the difference between these two: `error_messages` displays all errors for the form, while `error_messages_for` displays the errors for a given model.

Using the methods is easy:

```
<% form_for(@product) do |f| %>
  <%= f.error_messages %>
    <p>
    <!-- different form tags -->
    </p>
<% end %>
```

will display all the error messages: either default ones or the ones you have specified inside your model (`:message => "Only integers allowed"`). `error_messages_for` is used in a similar manner. Then it display the error message for the field that matches the given name (`:name` in this case)

```
<%= error_messages_for :name %>
```

You can also customize the error messages that are displayed in a box, even more:

```
<%= f.error_messages :header_message => "Invalid product!",
:message => "Correct your entries", :header_tag => :h6 %>
```

Advanced Forms

Until now we mainly worked with form elements that have a `_tag` at the end of the name. When working with a model (e.g. you want to update your data or create a new data set) you want to use the form elements that are more suited for the job. There are no new elements to learn we just use the one we already know but do not add the _tag at the end (compare `form_tag` and `form_for`).

Let's asume we want to create a new Product via a form:

We have a controller that has an action... `products_controller.rb`

```
#...
def new
  @product= Product.new
end
#...
```

... and a view (`view/products/new.html.erb`) that works with our `@product` instance:

```
<% form_for :product, @product, :url=>{:action => "create"} do
 |f| %>
  <%= f.text_field :name %>
  <%= f.text_field :price %, :size => 10 >
  <%= f.text_field :amount%, :size =>5 >
  <%= submit_tag "New Product" %>
<% end %>
```

will create HTML similar to:

```
<form action="/products" class="new_product" id="new_product"
method="post">
 <div style="margin:0;padding:0">
 <input name="authenticity_token" type="hidden"
value="bgqa1sbwcC5J/tIpPtJjX/3slveFNJg3WZntSOyHT4g=" />
 </div>
 <input id="product_name" name="post[name]" size="30" type="text"
/>
 <input id="product_price" name="post[price]" size="10" type="text"
/>
 <input id="product_amount" name="post[amount]" size="5" type="text"
 />
 <input id="product_submit" name="commit" type="submit"
value="Update" />

</form>
```

Let's inspect the code: `form_for` has some code on his side, `:product` references the name of the model that we want to use and `@product` is the instance of the object itself (in this case, it's going to be a new product). We loop or "yield" through the `form_for` object with the variable *f*. The rest uses code you are already familiar with. We use the action "create" to handle the creation of the file.

Restful Resources

The methods shown above may not be exactly what you will see in other Rails applications. Most of them will use RESTful resources. This gives Rails the option to decide what to do with the data. This makes the `form_for` method a lot easier and more readably:

```
form_for :product, @product, :url=>{:action => "create"}
```

turns into

```
form_for (@product)
```

That is a lot better, isn't it? Now the `form_for` tag looks the same, no matter if you are creating a new product or if you update an existing one. Rails is smart enough to recognize the action and provides us with the correct HTML attributes (see `<form action="/products" class="new_product" id="new_product" method="post">`)

To learn more about routing and REST be sure to take a look at Routing.

Using select boxes with a model

We already learnt how to create select boxes with built in helpers. But you may also want to display the contents of a model inside a select box. To show how this is done, we want to select categories from the database, so the user can select the according one:

```
<%=f.collection_select :category, :id, Category.all, :name,
:name.downcase %>
```

If you have some categories inside your database the HTML may look similar to

```
<select id="category_id" name="category[id]">
 <option value="cds">CDs</option>
 <option value="books">Books</option>
 <option value="dvds">DVDs</option>
</select>
```

Notice the `.downcase`: this will write the HTML values lowercase, if your database needs to work with it that way. This is just a starting point for your work. To see more examples and explanations, a good start is the official API (http://api.rubyonrails.org/classes/ActionView/Helpers/FormOptionsHelper.html)

Uploads

The following paragraph should give you just a quick overview on the topic of uploading data with forms. For the more advanced purposes you might want to consider using widely known gems that handle the job. Take a look at the Resources page to find good starting points for your search. Or take a look at the official Rails guide to see a sample action (http://guides.rubyonrails.org/form_helpers.html#uploading-files)

As with everything you want to upload via a form, you need to set the form to support "`multipart/form-data`". If you want to upload, e.g a picture for our products, you can use these ways: Here, you will need to write an action inside the controller that handles the upload of the file on the server:

```
<% form_tag({:action => :upload}, :multipart => true) do %>
  <%= file_field_tag 'picture' %>
<% end %>
```

For a form bound to a model we can use the already known `form_for tag` (this allows you to save e.g. the image name inside your database)

```
<% form_for @person, :html => {:multipart => true} do |f| %>
  <%= f.file_field :picture %>
<% end %>
```

Custom Helpers

Rails comes with a wide variety of standard view helpers. Helpers provide a way of putting commonly used functionality into a method which can be called in the view. Helpers include functionality for rendering URLs, formatting text and numbers, building forms and much more.

Custom Helpers

Custom helpers for your application should be located in the `app/helpers` directory.

Application Helper

The file

```
app/helpers/application.rb
```

contains helpers which are available to all views.

Controller Helpers

By default other helpers are mixed into the views for based on the controller name. For example, if you have a ProjectsController then you would have a corresponding ProjectsHelper in the file

```
app/helpers/projects_helper.rb
```

Example

The following is an example of an Application Helper. The method title will be available to all views in the application. Methods added to this helper will be available to all templates in the application.

```
module ApplicationHelper
    def title
      t = 'My Site'
      t << ": #{@title}" if @title
      t
    end
  end
```

ActionController - The Controller

Introduction

Now that you worked your way though Models and Views, it's time to get to the last part of MVC: the Controller. Your controller creates the proper data for the view and handles some logic structures. In order to fully understand the whole framework, you should also read about "Routing" to fully understand how things work together.

Actions

Actions are methods of your controller which respond to requests. For example:

```
class PeopleController < ApplicationController
      def index
        @people = Person.find(:all)
      end
      def show
        @people= Person.find(params[:id])
      end
    end
```

In this example the *people* controller has two actions: `index` and `show`. The action called `index` is the default action which is executed if no action is specified in the URL. For example:

```
http://localhost:3000/people
```

The `index` action can also be called explicitly:

```
http://localhost:3000/people/index
```

The show action must be called explicitly unless a route has been set up (routing will be covered later):

```
http://localhost:3000/people/show/1
```

In the example above the number 1 will be available in params[:id]. This is because the default route is:

```
map.connect :controller/:action/:id
```

This indicates that the first part of the path is the controller name, the second part the action name and the third part is the ID. A detailed explanation of routes is provided in its own chapter.

Parameters

Remember the "rendering and redirecting" example you've seen when learning about the view?

```
def update
    @product= Product.find(params[:id])

    if @product.update_attributes(params[:name])
      redirect_to :action => 'index'
    else
      render :edit
    end
end
```

You know what the differences between `render` and `redirect_to is`. Now we want to take a look at WHAT gets updates. This is determined by given the "update_attributes" method what we want to update: `params[:name]`. In this example, the data is likely to come from an HTML-Form (therefore a POST request). You can also work with GET requests in the same manner. Rails will handle both requests by using the params command.

When using e.g. check boxes you are likely to get more then one set of data for the same attribute. For example `option[]=1&option[]=2&option[]=3`. As with the example above, you can work with `params[:ids]` to get access to these settings. Your options will look like `options => {'1','2','3'}`

Session

For a technical explanation of a Session take a look at the wikipedia article about Sessions (http://en.wikipedia.org/wiki/Session_(computer_science))

In Rails you have some options to store the session. Most of the time you want to store the session on the server, but with security-relevant data, you might want to consider storing the session inside a database. To change the session storage, edit `config/initializers/session_store.rb` and be sure to read on the RoR Website (http://guides.rubyonrails.org/security.html) carefully.

Work with your session

As with the parameters, Rails provides a simple way of accessing your session. Consider following example:

```
def show_details
  #we may use this inside a user-specific action
  User.find(session[:current_user_id])
end
```

As you can see, you access the session in a similar way to the parameters. Storing a session isn't much more complicated:

```
def index
  #we have some code here to get the user_id of a specific (logged-in)
user
  session[:current_user_id] = id
end
```

To destroy the session, just assign it a nil-value

```
session[:current_user_id] = nil
```

Displaying a Flash-message

Flashes are very special and useful part of a session. You may have already found it in one of the view files. Here is how they work: As said, Flashes are special. They exist only once and are destroyed after each request. Flashes are useful to display error messages or notices to the user (e.g. when he tries to log in or if his request resulted in an error)

Inside an action flashes can be used similar to:

```
def check
  #code that does some validation
  flash[:notice] = "Successfull logged in"
end
```

Inside the view you can access it like:

```
<% if flash[:notice] -%>
    <%= flash[:notice] %>
<% end -%>
<!-- maybe some HTML-Code -->
<% if flash[:warning] -%>
    <%= flash[:warning] %>
<% end -%>
```

As you can see from the example above you are not limited to a single flash. You can access multiple flashes by their name you have defined inside the controller.

Cookies

Of course you can work with cookies inside your Rails application. This is, again very similar to working with parameters or sessions. Consider the following example:

```
def index
  #here we want to store the user name, if the user checked the
"Remember me" checkbox with HTML attribute name="remember_me"
  #we have already looked up the user and stored its object inside the
object-variable "@user"
 if params[:remember_me]
   cookies[:commenter_name] = @user.name
 else # Delete cookie for the commenter's name cookie, if any
  cookies.delete(:commenter_name)
 end
end
```

Routing

Because Routing is such an important part of Rails we dedicated a whole chapter to Routing (even though they are part of ActionView).

For example, if you want to display the product with the id 4, you will use a link similar to products/4. So Rails will figure out, that you want to show the information that belongs to the product with the id 4.

Routing also works when you want to link somewhere from one point of your application to another point. If you want to go back from the products view to the index overview that displays all products, you may place something like this in your view:

```
<%= link_to 'Back', products_path %>
```

When writing your own routes inside `routes.rb`, keep in mind, the lower the route is inside your file, the less priority it has.

Understanding Routes and Routing

RESTful routes

RESTful routes are the default routes in Rails. To get a more detailed technical view on REST, check out the Wikipedia (http://en.wikipedia.org/wiki/Representational_State_Transfer) article.

Basically REST provides a way of communication inside your application and all requests that exist from external sources (just as a browser request). To understand these principles better, take a look the following table:

HTTP Verb	URL	Controller	Action	used for
GET	/products	Product	index	display all products in an overview
GET	/products/new	Product	new	return an HTML form for creating a new product
POST	/products	Product	create	create a new product
GET	/products/1	Product	show	display a specific product
GET	/products/1/edit	Product	edit	return an HTML form for editing a product
PUT	/products/1	Product	update	update a specific product
DELETE	/products/1/	Product	destroy	delete a specific product

As you can see, all the actions of REST are already in our scaffolded controller. Keep in mind that RESTfoul routes reference a single object (products in this case). These 7 actions would result in a single route inside `routes.rb`:

```
map.resources :products
```

With a RESTful resource, it's easy to link the different views together or link to a specific view. With REST, Rails provides us some helpers to get to our desired location:

- products_url & products_path => redirects us to the index overview and edit view for our products (note the plural)
- new_product_url & new_product_path => will lead as to the form that creates a new product
- edit_product_url & edit_photo_path => provides us with an edit-form for a specific product
- product_url & product_path => is responsible for showing, deleting und updating a product

While *_path will create a relative path to, *_url provides the whole URL

- _path => /products

- _url => http://localhost/products

You will very likely need more than one REST route. You can easily write multiple REST routes into a single:

```
map.resources :products, :categories, :customers
```

You will come across many similar constructs like our products-category relation:

```
class Category < ActiveRecord::Base
  has_many :products
end

class Product < ActiveRecord::Base
  belongs_to :categories
end
```

HTTP Verb	URL	Controller	Action	used for
GET	/products/1/categories	Category	index	will show you an overview of the categories for product with the id 1
GET	/products/1/categories/new	Category	new	will give you an HTML form to create a new category for product with id 1
POST	/products/1/categories	Category	create	will create a new category for product with id 1
GET	/products/1/categories/1	Category	show	shows you the categories with id 1 that belongs to your product with id 1
GET	/products/1/categories/1/edit	Category	edit	gives you an HTML form to edit your category with id 1 that belongs to product with id 1
PUT	/products/1/categories/1	Category	update	updates category with id 1 that belongs to product with id 1
DELETE	/products/1/categories/1	Category	destroy	deletes category with id 1 that belongs to product with id 1

As with resources that are not nested, you will be able to access all *_url and *_path helpers e.g. `products_categories_path` or `new_product_category_url`

These path need to be present in your `routes.rb` in an similar maner to your model:

```
map.resources :products, :has_many => :categories
```

if you need to add more than association, put these in []

which is the same as the following route, only shorter

```
map.resources :magazines do |magazine|
  magazine.resources :ads
end
```

With this way, you can nest as many resources as you want, but you should try to keep the level of nesting as low as possible. So one nested resource is ok, but try to avoid 2 or more. To avoid these problems, we can use "shallow nesting"

If we want to add supplier for specific categories, we may end up with something like

```
map.resources :publishers, :shallow => true do |publisher|
  publisher.resources :magazines do |magazine|
    magazine.resources :photos
  end
end
```

or in short:

```
map.resources :products, :has_many => { :categories=> :suppliers
}, :shallow => true
```

There are many more advanced features for routing. More infos and instructions can be found in the official Rails guides (http://guides.rubyonrails.org/routing.html#controller-namespaces-and-routing).

Regular Routes

Even though RESTful routes are the encouraged way to go, you can also use regular routes inside your application. When working with regular routes, you give Rails some keywords and it will map the proper path for you. One of these default routes is already inside your `routes.rb` (this time we don't use `.resources` but `.connect`)

```
map.connect ':controller/:action/:id'
```

will for example be a browser request similar to `products/show/2`

If you are familiar with other languages that focus an the web, you may wonder how query strings inside the URL are handles: Rails automatically provides these parameters inside the params hash

So if you take the example above and add `products/show/2?category=3`, we would be able to access the category id with `params[:category_id]`.

See also

The API with lots of examples (http://api.rubyonrails.org/classes/ActionController/Routing.html)

ActiveSupport

Active Support is a collection of various utility classes and standard library extensions that were found useful for Rails. All these additions have hence been collected in this bundle as way to gather all that sugar that makes Ruby sweeter.

ActionMailer

Note: The following documentation was taken directly from the Rails API documentation (http://api.rubyonrails.com/classes/ActionMailer/Base.html)

ActionMailer allows you to send email from your application using a mailer model and views.

Model

To use ActionMailer, you need to create a mailer model.

```
script/generate mailer Notifier
```

The generated model inherits from `ActionMailer::Base`. Emails are defined by creating methods within the model which are then used to set variables to be used in the mail template to change options on the mail, or to add attachments.

Examples:

```
class Notifier < ActionMailer::Base
  def signup_notification(recipient)
    recipients recipient.email_address_with_name
```

```
    from        "system@example.com"
    subject     "New account information"
    body        "account" => recipient
  end
end
```

Mailer methods have the following configuration methods available.

- `recipients:` Takes one or more email addresses. These addresses are where your email will be delivered to. Sets the To: header.
- `subject:` The subject of your email. Sets the Subject: header.
- `from:` Who the email you are sending is from. Sets the From: header.
- `cc:`Takes one or more email addresses. These addresses will receive a carbon copy of your email. Sets the Cc: header.
- `bcc:` Takes one or more email address. These addresses will receive a blind carbon copy of your email. Sets the Bcc: header.
- `sent_on:` The date on which the message was sent. If not set, the header wil be set by the delivery agent.
- `content_type:` Specify the content type of the message. Defaults to text/plain.
- `headers:` Specify additional headers to be set for the message, e.g. headers 'X-Mail-Count' => 107370.

The body method has special behavior. It takes a hash which generates an instance variable named after each key in the hash containing the value that the key points do.

So, for example, body "account" => recipient would result in an instance variable @account with the value of recipient being accessible in the view.

Mailer Views

Like ActionController, each mailer class has a corresponding view directory in which each method of the class looks for a template with its name. To define a template to be used with a mailing, create an .rhtml file with the same name as the method in your mailer model. For example, in the mailer defined above, the template at

```
app/views/notifier/signup_notification.rhtml
```

would be used to generate the email.

Variables defined in the model are accessible as instance variables in the view.

Emails by default are sent in plain text, so a sample view for our model example might look like this:

```
Hi <%= @account.name %>,
Thanks for joining our service! Please check back often.
```

Sending HTML Mail

To send mail as HTML, make sure your view (the .rhtml file) generates HTML and set the content type to html.

```
class MyMailer < ActionMailer::Base
    def signup_notification(recipient)
      recipients recipient.email_address_with_name
      subject    "New account information"
      body       "account" => recipient
      from       "system@example.com"
      content_type "text/html"   #Here's where the magic happens
    end
```

```
  end
```

Multipart Mail

You can explicitly specify multipart messages:

```
class ApplicationMailer < ActionMailer::Base
    def signup_notification(recipient)
      recipients      recipient.email_address_with_name
      subject         "New account information"
      from            "system@example.com"

      part :content_type => "text/html",
        :body => render_message("signup-as-html", :account =>
recipient)

      part "text/plain" do |p|
        p.body = render_message("signup-as-plain", :account =>
recipient)
        p.transfer_encoding = "base64"
      end
    end
  end
```

Multipart messages can also be used implicitly because ActionMailer will automatically detect and use multipart templates where each template is named after the name of the action, followed by the content type. Each such detected template will be added as separate part to the message.

For example, if the following templates existed:

- `signup_notification.text.plain.rhtml`
- `signup_notification.text.html.rhtml`
- `signup_notification.text.xml.rxml`
- `signup_notification.text.x-yaml.rhtml`

Each would be rendered and added as a separate part to the message, with the corresponding content type. The same body hash is passed to each template.

Attachments

Attachments can be added by using the attachment method.

Example:

```
class ApplicationMailer < ActionMailer::Base
    # attachments
    def signup_notification(recipient)
      recipients      recipient.email_address_with_name
      subject         "New account information"
      from            "system@example.com"

      attachment :content_type => "image/jpeg",
        :body => File.read("an-image.jpg")
```

```
    attachment "application/pdf" do |a|
      a.body = generate_your_pdf_here()
    end
  end
end
```

Configuration Options

These options are specified on the class level, like ActionMailer::Base.template_root = "/my/templates"

- `template_root:` template root determines the base from which template references will be made.
- `logger:` the logger is used for generating information on the mailing run if available. Can be set to nil for no logging. Compatible with both Ruby's own Logger and Log4r loggers.
- `server_settings:` Allows detailed configuration of the server:
  - `:address` Allows you to use a remote mail server. Just change it from its default "localhost" setting.
  - `:port` On the off chance that your mail server doesn't run on port 25, you can change it.
  - `:domain` If you need to specify a HELO domain you can do it here.
  - `:user_name` If your mail server requires authentication, set the username in this setting.
  - `:password` If your mail server requires authentication, set the password in this setting.
  - `:authentication` If your mail server requires authentication you need to specify the authentication type here. This is a symbol and one of :plain, :login, :cram_md5
- `raise_delivery_errors:` whether or not errors should be raised if the email fails to be delivered.
- `delivery_method:` Defines a delivery method. Possible values are :smtp (default), :sendmail, and :test. Sendmail is assumed to be present at "/usr/sbin/sendmail".
- `perform_deliveries:` Determines whether deliver_* methods are actually carried out. By default they are, but this can be turned off to help functional testing.
- `deliveries:` Keeps an array of all the emails sent out through the Action Mailer with delivery_method :test. Most useful for unit and functional testing.
- `default_charset:` The default charset used for the body and to encode the subject. Defaults to UTF-8. You can also pick a different charset from inside a method with @charset.
- `default_content_type:` The default content type used for the main part of the message. Defaults to "text/plain". You can also pick a different content type from inside a method with @content_type.
- `default_mime_version:` The default mime version used for the message. Defaults to nil. You can also pick a different value from inside a method with @mime_version. When multipart messages are in use, @mime_version will be set to "1.0" if it is not set inside a method.
- `default_implicit_parts_order:` When a message is built implicitly (i.e. multiple parts are assembled from templates which specify the content type in their filenames) this variable controls how the parts are ordered. Defaults to ["text/html", "text/enriched", "text/plain"]. Items that appear first in the array have higher priority in the mail client and appear last in the mime encoded message. You can also pick a different order from inside a method with @implicit_parts_order.

Examples

This section is a collection of useful Rails examples.

Step By Step

How to Add a New Table

```
script/generate model <Name>
```

Generate the empty model and migration file.

```
vi db/migrate/XXX_create_<Name>.rb
```

Add columns to the table.

```
rake db:migrate
```

Migrates the data level - that is - creates the new database table.

```
vi app/models/<Name>.rb
```

Define the validations, sizes, etc.

```
vi test/unit/<Name>_test.rb
```

Define the unit tests that exercises the model validations.

If there will be a controller (and views) associated with this Model:

```
script/generate controller <Name> <action_one>
<action_two> ...
```

Creates the controller and creates a view for each action.

Find

The find method of ActiveRecord is documented in the Rails API manual (http://api.rubyonrails.org/classes/ActiveRecord/Base.html#M001376)

pet = Pet.find(pet_id) Find record by **id** (an integer). **Note:** Returns one object.

pets = Pet.find(:first, :conditions => ["owner_id = ?", owner_id]) - returns the **first** matching record. [Note: Returns one object.]

pets = Pet.find(:all, :conditions => ["owner_id = ?", owner_id]) - find all records with a given **field value**. [Notes: 1. Returns an array of objects. Check for no records found with: `pets.empty?`. 2.`:conditions =>` supplies an SQL fragment used with *`WHERE *]`*

pets = Pet.find(:all, :conditions => ["owner_id = ? AND name = ?", owner_id, name]) - find all records matching **multiple field values**. [Note: `OR` also works.]

pets = Pet.find(:all, :conditions => ["name LIKE ?", "Fido%"]) - find all records **matching a pattern**. Wild cards are `%` for zero or more of any character, and `_` for any single character. To escape a wild card use `\%` or `_`. The reference from MySQL (http://dev.mysql.com/doc/refman/5.0/en/string-comparison-functions.html#operator_like) for LIKE will help. On the MySQL Regex website (http://dev.mysql.com/doc/refman/5.0/en/regexp.html) you will find examples for using REGEX.

pets = Pet.find(:all, :order => 'name') - find everything and **sort result** by name.

pets = Pet.find(:all, :limit => 10, :conditions => ["owner_id = ?", owner_id]) - returns no more than the number of rows specified by **:limit**.

pets = Pet.find(:all, :offset => 50, :limit => 10) - uses **offset** to skip the first 50 rows.

Rake

Migrations

$ rake db:migrate - migrate to latest level by executing scripts in `<app>/db/migrate`. **Note:** Migration scripts are created by `script/generate model <mod-name>`

Testing

$ rake - run all tests.

$ rake test:functionals - run the functional tests, which test the controllers.

$ rake test:units - run the unit tests, which test the models.

$ test/functional/<name>_controller_test.rb - run one functional test.

Documentation

$ rake doc:app - generate Ruby Docs (http://rdoc.sourceforge.net/doc/index.html) for the application. Docs are placed at `<app>/doc/app/index.html`.

Clean Up

$ rake log:clear - delete all logs.

$ rake tmp:clear - delete temporary files.

Server

$ script/server - start the web server for this app. By default, the server is running in development mode. By default, it will be accessible at web address: http://localhost:3000/

$ RAILS_ENV=test script/server - start the web server in Test Mode.

$ script/server -e test - start the web server in Test Mode.

$ script/server -e production - start the web server in Production Mode (more caching, etc.).

Fixing Errors

can't convert Fixnum to String

some_number.to_s - every Fixnum has method `.to_s` to convert it to a String.

Shell Commands

Certain useful shell commands that I'm always trying to remember:

find . -exec grep 'hello' {} \; -print - run grep, searching for 'hello', on every file in the tree starting with the current directory.

tar -cvzf archive.tgz <targ_directory> - tar up directory and compress it with gzip.

Other Resources

There is a wide variety of documentation sources for Rails, including books, websites, weblogs and much more.

Plugins

- GITHub (http://github.com/search?q=rails)

  GITHub offers loads of plug-ins. Rails itself is also hosted there. Be sure to take a good look at them when searching for a specific plug-in or gem.

- popular Plugins (http://wiki.rubyonrails.org/#popular_plugins)

  Take at look at this list with pretty popular Rails plug-ins when searching for more common tasks

A large portion of the power of Rails comes from the wide range of plugins which are available for download.

Websites

- Rails API Documentation (http://api.rubyonrails.com/)
- Rails Wiki (http://wiki.rubyonrails.com/)

  The official wiki for Rails, not yet complete, but it gets better all the time

- Ruby on Rails plugin directory (http://www.railslodge.com/)
- Learning Ruby on Rails (http://www.yoyobrain.com/cardboxes/preview/863)
- Rails Documentation (http://rubyonrails.org/documentation)

  The official ROR site with everything you need to get started: guides, wikis and the very well documented API

- Videocasts for Rails (http://railscasts.com/)

  Very popular site with advanced and many in-sight video casts. Be sure to check this site out. It offers great content and well presented videos!

- Rails for PHP (http://railsforphp.com/)

  If you have problems wrapping your mind around some of the Ruby commands nad have experience in PHP, this site will feel like heaven. Most of the most often used PHP functions are listed there with the proper Ruby/Ruby on Rails equivalent

Books

- Agile Web Development with Ruby on Rails

Development Tools

- Rails Editor (http://e-texteditor.com/)

  Are you jealous of the great editor used on Railscasts? Not working an a Mac? This is the program for you: features the same functionality, look&feel and provides excellent Rails support. [Commercial - Windows]

- Open Source Development Suite Aptana (http://aptana.com/rails)

  comes with nice but complex Rails support, all Rake tasks are integrated into this IDE and you can easily switch between matching [Open Source and Commercial - All platforms]

- Editor for Rails (http://www.jetbrains.com/)

  Highly featured editor with rather new Rails support [Commercial - All plattforms]

- All-in-one IDE NetBeans (http://www.netbeans.org/features/ruby/index.html)

  offers Rails support out of the box. Support for many other languages and many plug-ins [Free - All Platforms]

- Rails and Eclipse (http://radrails.sourceforge.net/)

  If you want to work with eclipse, get RadRails. Note that the last update was submitted 2006. If you need an eclipse-like environment, try Aptana. It is built upon the eclipse platform.

Weblogs

Weblogs are one of the best ways to find out what you can do with rails - and if you know what you're looking for google can help you find them. That said, there are quite a few that are worth reading regularly.

- Ryan's Scraps (http://ryandaigle.com/)

  Insights into the framework from a core team member. Covers features that are new in edge.

- Nuby on Rails (http://nubyonrails.com/)

  Centers on the web design aspects of rails.

Mailing List

The Ruby on Rails Talk mailing list is a good place to ask questions: RubyOnRails-Talk Mailing List (http://groups.google.com/group/rubyonrails-talk)

Freenode ROR Channel

The Freenode IRC channel for Ruby on Rails is a good place to talk Rails with other developers. Please try to keep the signal-to-noise ratio at a minimum though: RubyOnRails IRC Channel (irc://irc.freenode.net/rubyonrails)

Article Sources and Contributors

Ruby Programming *Source*: http://en.wikibooks.org/w/index.php?oldid=1619046 *Contributors*: Adrignola, Briand, Dallas1278, Damien Karras, Darklama, Derbeth, Dysprosia, Effeietsanders, Eisel98, Elizabeth Barnwell, EvanCarroll, Herraotic, Hyad, Iamunknown, Jguk, Jk33, Keagan, Krischik, Marburg, Mehryar, Mkn, OinkOink, Orderud, Panic2k4, Pavan, QuiteUnusual, Ramir, Ravichandar84, Shirock, Sjc, Supriya kunjeer, Valters, Wantless, Withinfocus, Yath, Yuuki Mayuki, 60 anonymous edits

Ruby Programming/Overview *Source*: http://en.wikibooks.org/w/index.php?oldid=1550394 *Contributors*: Briand, Damien Karras, Georgesawyer, Jedediah Smith, Jguk, Rdnk, Scientus, Vovk, Withinfocus, Yath, 11 anonymous edits

Ruby Programming/Installing Ruby *Source*: http://en.wikibooks.org/w/index.php?oldid=1721388 *Contributors*: BiT, Briand, Eddy264, Gfranken, Jguk, Withinfocus, Yath, 7 anonymous edits

Ruby Programming/Ruby editors *Source*: http://en.wikibooks.org/w/index.php?oldid=1676751 *Contributors*: Briand, CAJDavidson, Damien Karras, Huw, Iron9light, Jim Mckeeth, Knudvaneeden, Krischik, Lhbts, Nbeyer, Snyce, 26 anonymous edits

Ruby Programming/Notation conventions *Source*: http://en.wikibooks.org/w/index.php?oldid=599024 *Contributors*: Briand

Ruby Programming/Interactive Ruby *Source*: http://en.wikibooks.org/w/index.php?oldid=724569 *Contributors*: Briand, Jguk, Sjc, Withinfocus, Yath, 4 anonymous edits

Ruby Programming/Mailing List FAQ *Source*: http://en.wikibooks.org/w/index.php?oldid=1574777 *Contributors*: Adrignola, 3 anonymous edits

Ruby Programming/Hello world *Source*: http://en.wikibooks.org/w/index.php?oldid=1633687 *Contributors*: Briand, Damien Karras, Jguk, Quilz, Sjc, Withinfocus, Yuuki Mayuki, 11 anonymous edits

Ruby Programming/Strings *Source*: http://en.wikibooks.org/w/index.php?oldid=1093022 *Contributors*: Briand, Damien Karras, 1 anonymous edits

Ruby Programming/Alternate quotes *Source*: http://en.wikibooks.org/w/index.php?oldid=1093030 *Contributors*: Briand, Damien Karras

Ruby Programming/Here documents *Source*: http://en.wikibooks.org/w/index.php?oldid=869201 *Contributors*: Briand, 2 anonymous edits

Ruby Programming/ASCII *Source*: http://en.wikibooks.org/w/index.php?oldid=1564234 *Contributors*: Briand, DavidCary, 1 anonymous edits

Ruby Programming/Introduction to objects *Source*: http://en.wikibooks.org/w/index.php?oldid=854422 *Contributors*: Briand, Lhbts

Ruby Programming/Ruby basics *Source*: http://en.wikibooks.org/w/index.php?oldid=1651769 *Contributors*: Ahy1, Benjamin Meinl, Briand, Dasch, Jeffq, Jguk, Marburg, Scopsowl, Withinfocus, 10 anonymous edits

Ruby Programming/Data types *Source*: http://en.wikibooks.org/w/index.php?oldid=1706913 *Contributors*: Archaeometallurg, Gamache, IanVaughan, Ryepdx, 11 anonymous edits

Ruby Programming/Writing methods *Source*: http://en.wikibooks.org/w/index.php?oldid=1649985 *Contributors*: Briand, IanVaughan, Jguk, Joti, Paul.derry, Withinfocus, 12 anonymous edits

Ruby Programming/Classes and objects *Source*: http://en.wikibooks.org/w/index.php?oldid=1467994 *Contributors*: IanVaughan, 7 anonymous edits

Ruby Programming/Exceptions *Source*: http://en.wikibooks.org/w/index.php?oldid=1648307 *Contributors*: Adrignola, Θεόφιλε, 3 anonymous edits

Ruby Programming/Unit testing *Source*: http://en.wikibooks.org/w/index.php?oldid=1688425 *Contributors*: Briand, Jguk, LuisParravicini, Marburg, Withinfocus, 12 anonymous edits

Ruby Programming/RubyDoc *Source*: http://en.wikibooks.org/w/index.php?oldid=1139096 *Contributors*: Jguk, LuisParravicini, Marburg, Withinfocus, 6 anonymous edits

Ruby Programming/Syntax *Source*: http://en.wikibooks.org/w/index.php?oldid=566309 *Contributors*: Jguk, Keagan, Withinfocus, 1 anonymous edits

Ruby Programming/Syntax/Lexicology *Source*: http://en.wikibooks.org/w/index.php?oldid=1679154 *Contributors*: Adrignola, Darklama, Jguk, Keagan, Munificent, RichardOnRails, Withinfocus, 15 anonymous edits

Ruby Programming/Syntax/Variables and Constants *Source*: http://en.wikibooks.org/w/index.php?oldid=1709548 *Contributors*: Adrignola, Darklama, Jguk, Keagan, Valters, Withinfocus, Yuuki Mayuki, 15 anonymous edits

Ruby Programming/Syntax/Literals *Source*: http://en.wikibooks.org/w/index.php?oldid=1539430 *Contributors*: Adrignola, Dasch, Dmw, Ghostzart, HenryLi, Jedediah Smith, Jguk, Karvendhan, Keagan, Mehryar, Valters, Withinfocus, Yuuki Mayuki, 25 anonymous edits

Ruby Programming/Syntax/Operators *Source*: http://en.wikibooks.org/w/index.php?oldid=1707069 *Contributors*: Adrignola, Derbeth, Jguk, Jordandanford, Mehryar, RichardOnRails, Srogers, Stiang, Valters, Withinfocus, 24 anonymous edits

Ruby Programming/Syntax/Control Structures *Source*: http://en.wikibooks.org/w/index.php?oldid=1633193 *Contributors*: Adrignola, Darklama, Fricky, Geoffj, Jguk, Raevel, Withinfocus, Yuuki Mayuki, Θεόφιλε, 25 anonymous edits

Ruby Programming/Syntax/Method Calls *Source*: http://en.wikibooks.org/w/index.php?oldid=1693384 *Contributors*: Adrignola, Bapabooiee, Darklama, Eshafto, Fhope, Jguk, QuiteUnusual, Scientes, Scientus, Valters, Withinfocus, 31 anonymous edits

Ruby Programming/Syntax/Classes *Source*: http://en.wikibooks.org/w/index.php?oldid=1738733 *Contributors*: Adrignola, Florian bravo, Formigarafa, Jguk, Joti, Kenfodder, Oleander, Rule.rule, Valters, Withinfocus, 64 anonymous edits

Ruby Programming/Reference *Source*: http://en.wikibooks.org/w/index.php?oldid=1288464 *Contributors*: Jguk, Mehryar, Withinfocus, 2 anonymous edits

Ruby Programming/Reference/Predefined Variables *Source*: http://en.wikibooks.org/w/index.php?oldid=1574811 *Contributors*: Adrignola, Eisel98

Ruby Programming/Reference/Predefined Classes *Source*: http://en.wikibooks.org/w/index.php?oldid=1574810 *Contributors*: Adrignola, EvanCarroll

Ruby Programming/Reference/Objects *Source*: http://en.wikibooks.org/w/index.php?oldid=1488337 *Contributors*: Adrignola, Cspurrier, Jguk, Withinfocus, 3 anonymous edits

Ruby Programming/Reference/Objects/Array *Source*: http://en.wikibooks.org/w/index.php?oldid=1488338 *Contributors*: Adrignola, Jguk, Withinfocus, 3 anonymous edits

Ruby Programming/Object/NilClass *Source*: http://en.wikibooks.org/w/index.php?oldid=1237900 *Contributors*: Ramac, 1 anonymous edits

Ruby Programming/Reference/Objects/Exception *Source*: http://en.wikibooks.org/w/index.php?oldid=1574780 *Contributors*: Adrignola, Eisel98

Ruby Programming/Reference/Objects/FalseClass *Source*: http://en.wikibooks.org/w/index.php?oldid=1574781 *Contributors*: Adrignola, Eisel98

Ruby Programming/Reference/Objects/IO/File/File::Stat *Source*: http://en.wikibooks.org/w/index.php?oldid=1488339 *Contributors*: Adrignola, Jguk, Withinfocus, 1 anonymous edits

Ruby Programming/Reference/Objects/Numeric *Source*: http://en.wikibooks.org/w/index.php?oldid=1574782 *Contributors*: Adrignola, Eisel98

Ruby Programming/Reference/Objects/Numeric/Integer *Source*: http://en.wikibooks.org/w/index.php?oldid=1574785 *Contributors*: Adrignola, 1 anonymous edits

Ruby Programming/Reference/Objects/Regexp *Source*: http://en.wikibooks.org/w/index.php?oldid=1488341 *Contributors*: Adrignola, Jguk, Vanivk, Withinfocus

Ruby Programming/Reference/Objects/String *Source*: http://en.wikibooks.org/w/index.php?oldid=1574805 *Contributors*: Adrignola, 1 anonymous edits

Ruby Programming/Reference/Objects/Symbol *Source*: http://en.wikibooks.org/w/index.php?oldid=1574806 *Contributors*: Adrignola, 1 anonymous edits

Ruby Programming/Reference/Objects/Time *Source*: http://en.wikibooks.org/w/index.php?oldid=1574807 *Contributors*: Adrignola, 1 anonymous edits

Ruby Programming/Reference/Objects/TrueClass *Source*: http://en.wikibooks.org/w/index.php?oldid=1574809 *Contributors*: Adrignola, Eisel98

Ruby Programming/Reference/Built-in Modules *Source*: http://en.wikibooks.org/w/index.php?oldid=1574779 *Contributors*: Adrignola, Mehryar

Ruby Programming/Built-in Modules *Source*: http://en.wikibooks.org/w/index.php?oldid=1357476 *Contributors*: Eisel98, Mehryar, 2 anonymous edits

Ruby Programming/GUI Toolkit Modules/Tk *Source*: http://en.wikibooks.org/w/index.php?oldid=1488333 *Contributors*: Adrignola, Darklama, Hagindaz, Jguk, Withinfocus, 4 anonymous edits

Ruby Programming/GUI Toolkit Modules/GTK2 *Source*: http://en.wikibooks.org/w/index.php?oldid=1617228 *Contributors*: Adrignola, Rbidegain

Ruby Programming/GUI Toolkit Modules/Qt4 *Source*: http://en.wikibooks.org/w/index.php?oldid=1574776 *Contributors*: Adrignola, 3 anonymous edits

Ruby Programming/XML Processing/REXML *Source*: http://en.wikibooks.org/w/index.php?oldid=1488349 *Contributors*: Adrignola, Jguk, Withinfocus, Yuuki Mayuki

Ruby on Rails/Print version *Source*: http://en.wikibooks.org/w/index.php?oldid=1534235 *Contributors*: At.fhj.itm

Image Sources, Licenses and Contributors

File:Ruby_logo.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Ruby_logo.png *License*: Creative Commons Attribution-Sharealike 2.5 *Contributors*: Yukihiro Matsumoto, Ruby Visual Identity Team

File:25%.svg *Source*: http://en.wikibooks.org/w/index.php?title=File:25%.svg *License*: Public Domain *Contributors*: Karl Wick

Image:Ruby - Introduction to objects - Variable reference.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Ruby_-_Introduction_to_objects_-_Variable_reference.png *License*: unknown *Contributors*: Briand

image:Ruby - Introduction to objects - Two variables.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Ruby_-_Introduction_to_objects_-_Two_variables.png *License*: unknown *Contributors*: Briand

image:Ruby - Introduction to objects - Two variables, modified object.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Ruby_-_Introduction_to_objects_-_Two_variables,_modified_object.png *License*: unknown *Contributors*: Briand

image:Ruby - Introduction to objects - Two variables, two objects.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Ruby_-_Introduction_to_objects_-_Two_variables,_two_objects.png *License*: unknown *Contributors*: Briand

image:Ruby - Introduction to objects - Orphaned object.png *Source*: http://en.wikibooks.org/w/index.php?title=File:Ruby_-_Introduction_to_objects_-_Orphaned_object.png *License*: unknown *Contributors*: Briand

Image:Mvc_rails.jpg *Source*: http://en.wikibooks.org/w/index.php?title=File:Mvc_rails.jpg *License*: Public Domain *Contributors*: At.fhj.itm

File:RoR_diagram_associations.jpg *Source*: http://en.wikibooks.org/w/index.php?title=File:RoR_diagram_associations.jpg *License*: Public Domain *Contributors*: At.fhj.itm

File:RoR_has_many_through.jpg *Source*: http://en.wikibooks.org/w/index.php?title=File:RoR_has_many_through.jpg *License*: Public Domain *Contributors*: At.fhj.itm

File:RoR has one through.jpg *Source*: http://en.wikibooks.org/w/index.php?title=File:RoR_has_one_through.jpg *License*: Public Domain *Contributors*: At.fhj.itm

License

Creative Commons Attribution-Share Alike 3.0 Unported
http://creativecommons.org/licenses/by-sa/3.0/