Prepared By Dr. N.THENMOZHI M.C.A., M.S., M.Phil., Ph.D.,

Govt, Arts College (Autonomous), Coimbatore-18

Department of Information Technology

OPEN SOURCE TOOLS (18MIT33C) -----II MSc

UNIT-V: Ruby on Rails: Welcome to Ruby –Conditions, methods, loops and blocks - classes and objects. Welcome to rails: Connecting to databases – working with databases.

Text Book : Steven Holzner, "Beginning ruby on rails", Wiley publishing, Inc, 2007.

5. INTRODUCTION

Ruby is the programming language you're going to be using, and Rails is the web application framework that will put everything online.

Ruby is a pure object-oriented programming language. It was created in 1993 by Yukihiro Matsumoto of Japan.

You can find the name Yukihiro Matsumoto on the Ruby mailing list at www.ruby-lang.org. Matsumoto is also known as Matz in the Ruby community.

What is Ruby?

Ruby is the successful combination of −

- Smalltalk's conceptual elegance,
- Python's ease of use and learning, and
- Perl's pragmatism.

Ruby is −

- A high-level programming language.
- Interpreted like Perl, Python, Tcl/TK.
- Object-oriented like Smalltalk, Eiffel, Ada, Java.

Why Ruby?

Ruby originated in Japan and now it is gaining popularity in US and Europe as well. The following factors contribute towards its popularity −

- Easy to learn
- Open source (very liberal license)
- Rich libraries
- Very easy to extend
- Truly object-oriented
- Less coding with fewer bugs
- Helpful community

Although we have many reasons to use Ruby, there are a few drawbacks as well that you may have to consider before implementing Ruby −

Performance Issues − Although it rivals Perl and Python, it is still an interpreted language and we cannot compare it with high-level programming languages like C or C++.

Threading model − Ruby does not use native threads. Ruby threads are simulated in the VM rather than running as native OS threads.

Ruby is "A Programmer's Best Friend".

Ruby has features that are similar to those of Smalltalk, Perl, and Python. Perl, Python, and Smalltalk are scripting languages. Smalltalk is a true object-oriented language. Ruby, like Smalltalk, is a perfect object-oriented language. Using Ruby syntax is much easier than using Smalltalk syntax.

Features of Ruby

- Ruby is an open-source and is freely available on the Web, but it is subject to a license.
- Ruby is a general-purpose, interpreted programming language.
- Ruby is a true object-oriented programming language.
- Ruby is a server-side scripting language similar to Python and PERL.
- Ruby can be used to write Common Gateway Interface (CGI) scripts.
- Ruby can be embedded into Hypertext Markup Language (HTML).
- Ruby has a clean and easy syntax that allows a new developer to learn very quickly and easily.
- Ruby has similar syntax to that of many programming languages such as C++ and Perl.
- Ruby is very much scalable and big programs written in Ruby are easily maintainable.
- Ruby can be used for developing Internet and intranet applications.
- Ruby can be installed in Windows and POSIX environments.
- Ruby support many GUI tools such as Tcl/Tk, GTK, and OpenGL.
- Ruby can easily be connected to DB2, MySQL, Oracle, and Sybase.
- Ruby has a rich set of built-in functions, which can be used directly into Ruby scripts.

Tools You Will Need

You will need a latest computer like Intel Core i3 or i5 with a minimum of 2GB of RAM (4GB of RAM recommended). You also will need the following software

- Linux or Windows 95/98/2000/NT or Windows 7 operating system.
- Apache 1.3.19-5 Web server.
- Internet Explorer 5.0 or above Web browser.
- Ruby 1.8.5

5.1. Welcome to Ruby

Install Ruby and Rails on Windows

Just follow these steps to install Ruby:

- Download the one-click installer for Ruby at http://rubyinstaller.rubyforge.org.
- Click the installer to install Ruby.

Select Start➪Run
Type **cmd** in the Open field and click OK.
Then, type the following at the command prompt:
gem install rails --include-dependencies

Install Ruby and Rails in Linux and Unix

Open a shell and type **ruby -v** at the prompt

if you have version 1.8.2 or later installed, you're all set. If you don't have Ruby installed, you can find pre-built versions for your Linux/Unix installation on the Internet, or you can build it from the source, which you can find at
http:// ruby-lang.org/en.

1. tar xzf ruby-1.8.4.tar.gz
2. cd ruby-1.8.4
3. ./configure
4. make
5. make test
6. sudo make install

You're also going to need Rails, which is most easily installed with RubyGems. To get RubyGems, go to

http://rubygems.rubyforge.org
and click the download link.
Then go to the directory containing the download in a shell and enter the following at the command prompt, updating rubygems-0.8.10.tar.gz to the most recent version of the download:

1. tar xzf rubygems-0.8.10.tar.gz
2. cd rubygems-0.8.10
3. sudo ruby setup.rb

All that's left is to use RubyGems to install Rails, which you can do this way:
sudo gem install rails --include-dependencies

Popular Ruby Editors

- If you are working on Windows machine, use text editor Notepad or Edit plus.
- VIM (Vi IMproved) is a very simple text editor. This is available on almost all Unix machines and now Windows as well. Otherwise, your can use your favorite vi editor to write Ruby programs.
- RubyWin is a Ruby Integrated Development Environment (IDE) for Windows.
- Ruby Development Environment (RDE) is also a very good IDE for windows users.

Interactive Ruby (IRb)

Interactive Ruby (IRb) provides a shell for experimentation. Within the IRb shell, you can immediately view expression results, line by line.

This tool comes along with Ruby installation so you have nothing to do extra to have IRb working.

Just type **irb** at your command prompt and an Interactive Ruby Session will start as given below

```
$irb
irb 0.6.1(99/09/16)
irb(main):001:0> def hello
irb(main):002:1> out = "Hello World"
irb(main):003:1> puts out
irb(main):004:1> end
nil
irb(main):005:0> hello
Hello World
nil
irb(main):006:0>
```

Getting Started with Ruby

Ruby is the language that is going to make everything happen. To work with Ruby, you need a text editor
Programs ➪ Accessories ➪ WordPad) or Notepad (Start ➪ Programs ➪ Accessories ➪ Notepad) in Windows.

Each Ruby program should be saved with the extension .rb, such as this first example, hello.rb,which displays a greeting from Ruby.

Example 5.1 Display a Message

1. Start your text editor and enter the following Ruby code:
   puts "Hello from Ruby."
2. Save the file as **hello.rb.** Make sure you save the file as a text file (select Text Document in the Save As Type drop-down), and if you are using Windows WordPad or Notepad, make sure you enclose the name of the file in quotes—"hello.rb"—before saving to prevent those editors from saving the file as hello.rb.rtf or hello.rb.txt.

3. Use Ruby to run this new program and see the results. Just enter the ruby command followed by the name of the program at the command line:

C:\rubydev>ruby hello.rb

Output

Hello from Ruby

5.1.1. Ruby - Syntax

Whitespace in Ruby Program

Whitespace characters such as spaces and tabs are generally ignored in Ruby code, except when they appear in strings. Sometimes, however, they are used to interpret ambiguous statements. Interpretations of this sort produce warnings when the -w option is enabled.

Example

a + b is interpreted as a+b (Here a is a local variable)
a +b is interpreted as a(+b) (Here a is a method call)

Line Endings in Ruby Program

Ruby interprets semicolons and newline characters as the ending of a statement. However, if Ruby encounters operators, such as +, −, or backslash at the end of a line, they indicate the continuation of a statement.

Ruby Identifiers

Identifiers are names of variables, constants, and methods. Ruby identifiers are case sensitive. It means Ram and RAM are two different identifiers in Ruby.Ruby identifier names may consist of alphanumeric characters and the underscore character (_).

Here Document in Ruby

"Here Document" refers to build strings from multiple lines. Following a << you can specify a string or an identifier to terminate the string literal, and all lines following the current line up to the terminator are the value of the string.

If the terminator is quoted, the type of quotes determines the type of the line-oriented string literal. Notice there must be no space between << and the terminator.

```
#!/usr/bin/ruby -w

print <<EOF
   This is the first way of creating
   here document ie. multiple line string.
EOF

print <<"EOF";                # same as above
   This is the second way of creating
   here document ie. multiple line string.
EOF

print <<`EOC`                 # execute commands
            echo hi there
            echo lo there
```

```
EOC

print <<"foo", <<"bar"  # you can stack them
        I said foo.
foo
        I said bar.
bar
```

This will produce the following result −

```
  This is the first way of creating
  her document ie. multiple line string.
  This is the second way of creating
  her document ie. multiple line string.
hi there
lo there
    I said foo.

            I said bar.
```

Checking the Ruby Documentation

What about documentation?

To handle the local version of the documentation, use the ri tool. Just enter **ri** at the command line, followed by the item you want help with, such as the puts method:

C:\rubydev>ri puts

In this example, the puts method of interest is part of the IO package, so request that documentation by entering **ri IO#puts** at the command line:

```
irb(main):001:0> puts “Hello from Ruby.”
```

When you press the Enter key, irb evaluates your Ruby and gives you the result:

```
C:\rubydev>irb
irb(main):001:0> puts “Hello from Ruby.”
Hello from Ruby.
```

Although you can create multi-line programs using irb, it’s awkward, so this book sticks to entering Ruby code in files instead. arguments, outputs a single record separator.

```
$stdout.puts(“this”, “is”, “a”, “test”)
The output is
this
is
a
test
```

Ruby Data types

Data types represents a type of data such as text, string, numbers, etc. There are different data types in Ruby:

- ✓ Numbers
- ✓ Strings
- ✓ Symbols
- ✓ Hashes
- ✓ Arrays
- ✓ Booleans

Working with Numbers in Ruby

Ruby has some great features for working with numbers. In fact, Ruby handles numbers automatically. There is no limit to the size of integers you can use a number like 12345678987654321. The floating-point numbers simply by using a decimal point like this: 3.1415, also give an exponent like this: 31415.0e-4. And you can give binary numbers by prefacing them with 0b (as in 0b1111), octal—base eight—numbers by prefacing them with a 0 (like this: 0355), and hexadecimal numbers—base 16—by prefacing them with 0x (such as 0xddff).

Ruby stores numbers in a variety of types. For example, integers are stored as the* Fixnum *type unless they become too large, in which case they are stored as the* Bignum *type. And floating-point numbers are stored as the* Float *type.

To get started with numbers in Ruby, follow these steps:
1. Enter this Ruby code in a new file:

```
puts 12345
puts 3.1415
puts 31415.0e-4
puts 12_345_678_987_654_321
puts 0xddff
```

2. Save the file as numbers.rb.
3. Use Ruby to run numbers.rb:

```
C:\rubydev>ruby numbers.rb
12345
3.1415
3.1415
12345678987654321
56831
```

Strings in Ruby

Ruby string object holds and manipulates an arbitary sequence of bytes, typically representing characters. They are created using **String::new** or as literals.

Quotes : Ruby string literals are enclosed within single and double quotes.

Example:

```
1. #!/usr/bin/ruby
2. puts 'Hello everyone'
3. puts "Hello everyone"
```

Output:

Hello everyone
Hello everyone

Use the %q (single quotes) or %Q (double quotes) inside the syntax instead quotes. Just use %q or %Q with a single character, and Ruby will add quotes until it sees that character again. For example,

You can also concatenate (join) strings together, using a +. For example, the expression "It " + "was "+ "too " + "Cary " + "Grant!" is the same as "It was too Cary Grant!".

1. Enter this Ruby code in a new file, strings.rb:

```
puts "Hello"
puts "Hello " + "there"
puts 'Nice to see you.'
puts %Q/How are you?/
puts %Q!Fine, and you?!
puts %q!I'm also fine, thanks.!
puts "I have to say, 'I am well.'"
puts "I'll also say, \"Things are fine.\""
```

1. Save the file as strings.rb.
2. Run strings.rb using Ruby to see the result:

C:\rubydev>ruby strings.rb

Hello
Hello there
Nice to see you.
How are you?
Fine, and you?
I'm also fine, thanks.
I have to say, 'I am well.'
I'll also say, "Things are fine."

Accessing string elements

You can access Ruby string elements in different parts with the help of square brackets []. Within square brackets write the index or string.

Example:

```
1. #!/usr/bin/ruby
2.  msg = "This is Testing Ruby String."
3.  puts msg["Testing"]
4.  puts msg["Ruby"]
5.  puts msg[0]
6.  puts msg[0, 2]
7.  puts msg[0..9]
8.  puts msg[0, msg.length]
9.  puts msg[-3]
```

Output:

Testing
Ruby

T
Th
This is Te
This is Testing Ruby String.
n

Multiline string

Writing multiline string is very simple in Ruby language. We will show three ways to print multi line string.

- ✓ String can be written within double quotes.
- ✓ The % character is used and string is enclosed within / character.
- ✓ In heredoc syntax, we use << and string is enclosed within word STRING.

Example:

```
1.  puts "
2.  A
3.  AB
4.  ABC
5.  ABCD"
6.
7.  puts %/
8.  A
9.  AB
10. ABC
11. ABCD/
12.
13. puts <<STRING
14. A
15. AB
16. ABC
17. ABCD
18. STRING
```

Output:

```
A
AB
ABC
ABCD

A
AB
ABC
ABCD
A
AB
ABC
ABCD
```

Variable Interpolation

Ruby variable interpolation is replacing variables with values inside string literals. The variable name is put between #{ and } characters inside string literal.

Example:

```
1.  #!/usr/bin/ruby
2.  country = "India"
3.  capital = "New Delhi"
4.  puts "#{capital} is the capital of #{country}."
```

Output:

New Delhi is the capital of India

Concatenating Strings

Ruby concatenating string implies creating one string from multiple strings. You can join more than one string to form a single string by concatenating them. There are four ways to concatenate Ruby strings into single string:

- ✓ Using plus sign in between strings.
- ✓ Using a single space in between strings.
- ✓ Using << sign in between strings.
- ✓ Using concat method in between strings.

Example:

```
1.  #!/usr/bin/ruby
2.    string = "This is Ruby Tutorial" + " from Book." + " Wish you all good luck."
3.  puts string
4.    string = "This is Ruby Tutorial" " from Book." " Wish you all good luck."
5.  puts string
6.   string = "This is Ruby Tutorial" << " from Book." << " Wish you all good luck."
7.  puts string
8.    string = "This is Ruby Tutorial".concat(" from Book.").concat(" Wish you all good luck.
    ")
9.  puts string
```

Output:

This is Ruby Tutorial from Book. Wish you all good luck.

This is Ruby Tutorial from Book. Wish you all good luck.

This is Ruby Tutorial from Book. Wish you all good luck.

This is Ruby Tutorial from Book. Wish you all good luck.

Freezing Strings

In most programming languages strings are immutable. It means that an existing string can't be modified, only a new string can be created out of them. In Ruby, by default strings are not immutable. To make them immutable, freeze method can be used.

Example:

```
1.  #!/usr/bin/ruby
2.  str = "Original string"
3.  str << " is modified "
```

```
4.  str << "is again modified"
5.   puts str
6.    str.freeze
7.    #str << "And here modification will be failed after using freeze method"
```

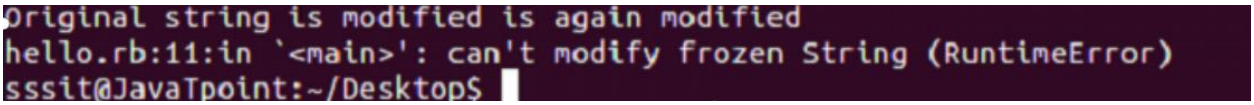
Output:

Original string is modified is again modified

In the above output, we have made the string immutable by using freeze method. Last line is commented as no string can't be modified any further.

By uncommenting the last line, we'll get an error as shown in the below output.

Output:



Comparing Strings

Ruby strings can be compared with three operators:

- ✓ With == operator : Returns true or false
- ✓ With eql? Operator : Returns true or false
- ✓ With casecmp method : Returns 0 if matched or 1 if not matched

Example:

```
1.  #!/usr/bin/ruby
2.    puts "abc" == "abc"
3.  puts "as ab" == "ab ab"
4.  puts "23" == "32"
5.    puts "ttt".eql? "ttt"
6.  puts "12".eql? "12"
7.    puts "Java".casecmp "Java"
8.  puts "Java".casecmp "java"
9.  puts "Java".casecmp "ja"
```

Output:

True
False
False
true
true
0
0
1

Ruby Variables

Ruby variables are locations which hold data to be used in the programs. Each variable has a different name. These variable names are based on some naming conventions. Unlike other programming languages, there is no need to declare a variable in Ruby. A prefix is needed to indicate it.

There are four types of variables in Ruby:

- ✓ Local variables
- ✓ Class variables
- ✓ Instance variables
- ✓ Global variables

Local variables

A local variable name starts with a lowercase letter or underscore (_). It is only accessible or have its scope within the block of its initialization. Once the code block completes, variable has no scope.

When uninitialized local variables are called, they are interpreted as call to a method that has no arguments.

Class variables

A class variable name starts with @@ sign. They need to be initialized before use. A class variable belongs to the whole class and can be accessible from anywhere inside the class. If the value will be changed at one instance, it will be changed at every instance.

A class variable is shared by all the descendents of the class. An uninitialized class variable will result in an error.

Example:

```
1.  #!/usr/bin/ruby
2.   class States
3.     @@no_of_states=0
4.     def initialize(name)
5.       @states_name=name
6.       @@no_of_states += 1
7.     end
8.     def display()
9.       puts "State name #@state_name"
10.    end
11.    def total_no_of_states()
12.      puts "Total number of states written: #@@no_of_states"
13.    end
14.  end
15.   # Create Objects
16.  first=States.new("Assam")
17.  second=States.new("Meghalaya")
18.  third=States.new("Maharashtra")
19.  fourth=States.new("Pondicherry")
20.   # Call Methods
21.  first.total_no_of_states()
22.  second.total_no_of_states()
23.  third.total_no_of_states()
24.  fourth.total_no_of_states()
```

In the above example, @@no_of_states is a class variable.

Output:

Total number of states written: 4

Total number of states written: 4
Total number of states written: 4
Total number of states written: 4

Instance variables

An instance variable name starts with a @ sign. It belongs to one instance of the class and can be accessed from any instance of the class within a method. They only have limited access to a particular instance of a class. They don't need to be initialize. An uninitialized instance variable will have a nil value.

Example:

```
1.  #!/usr/bin/ruby
2.    class States
3.     def initialize(name)
4.       @states_name=name
5.     end
6.     def display()
7.       puts "States name #@states_name"
8.     end
9.  end
10.   # Create Objects
11.  first=States.new("Assam")
12.  second=States.new("Meghalaya")
13.  third=States.new("Maharashtra")
14.  fourth=States.new("Pondicherry")
15.   # Call Methods
16.  first.display()
17.  second.display()
18.  third.display()
19.  fourth.display()
```

In the above example, @states_name is the instance variable.

Output:

States name Assam
States name Meghalaya
States name Maharashtra
States name Pondicherry

Global variables

A global variable name starts with a $ sign. Its scope is globally, means it can be accessed from any where in a program. An uninitialized global variable will have a nil value. It is advised not to use them as they make programs cryptic and complex. There are a number of predefined global variables in Ruby.

Example:

```
1.  #!/usr/bin/ruby
2.   $global_var = "GLOBAL"
3.  class One
4.   def display
```

```
5.          puts "Global variable in One is #$global_var"
6.        end
7.      end
8.      class Two
9.        def display
10.         puts "Global variable in Two is #$global_var"
11.       end
12.     end
13.       oneobj = One.new
14.     oneobj.display
15.     twoobj = Two.new
16.     twoobj.display
```

In the above example, @states_name is the instance variable.

Output:

Global variable in One is GLOBAL
Global variable in Two is GLOBAL

Summary

	Local	Global	Instance	Class
Scope	Limited within the block of initialization.	Its scope is globally.	It belongs to one instance of a class.	Limited to the whole class in which they are created.
Naming	Starts with a lowercase letter or underscore (_).	Starts with a $ sign.	Starts with an @ sign.	Starts with an @@ sign.
Initialization	No need to initialize. An uninitialized local variable is interpreted as methods with no arguments.	No need to initialize. An uninitialized global variable will have a nil value.	No need to initialize. An uninitialized instance variable will have a nil value.	They need to be initialized before use. An uninitialized global variable results in an error.

Storing Data in Variables

Ruby can store your data in *variables*, which are named placeholders that can store numbers, strings, and other data. You reference the data stored in a variable by using the variable's name. For example, to store a value of 34 in a variable named temperature, you assign that variable the value like this:

```
temperature = 34
puts temperature
```

Here's the result:

Rules for the names you can use in Ruby

A standard variable starts with a lowercase letter, *a* to *z*, or an underscore, _, followed by any number of *name characters*. A name character is a lowercase letter, an uppercase letter, a digit, or an underscore. And you have to avoid the words that Ruby reserves for itself.

Example

```
temperature = 36
puts “The temperature is “ + String(temperature) + “.”
temperature = temperature + 5
puts “Now the temperature is “ + String(temperature) + “.”
```

Output

The temperature is 36.
Now the temperature is 41.

Creating Constants

A *constant* holds a value that you do not expect to change, such as the value of pi:

```
PI = 3.1415926535
```

Symbols

Symbols are like strings. A symbol is preceded by a colon (:). For example,

:abcd

They do not contain spaces. Symbols containing multiple words are written with (_). One difference between string and symbol is that, if text is a data then it is a string but if it is a code it is a symbol.

Symbols are unique identifiers and represent static values, while string represent values that change.

Example:

```
irb(main):009:0> "string".object_id
=> 72164150
irb(main):010:0> "string".object_id
=> 71964380
irb(main):011:0> :symbol.object_id
=> 223448
irb(main):012:0> :symbol.object_id
=> 223448
irb(main):013:0>
```

In the above snapshot, two different object_id is created for string but for symbol same object_id is created.

Hashes

A hash assign its values to its keys. They can be looked up by their keys. Value to a key is assigned by => sign. A key/value pair is separated with a comma between them and all the pairs are enclosed within curly braces. For example,

{"Akash" => "Physics", "Ankit" => "Chemistry", "Aman" => "Maths"}

Example:

```
1.  #!/usr/bin/ruby
2.  data = {"Akash" => "Physics", "Ankit" => "Chemistry", "Aman" => "Maths"}
3.  puts data["Akash"]
4.  puts data["Aman"]
5.  puts data["Ankit"]
```

Output:

Physics
Maths
Chemistry

Ruby Hashes

A Ruby hash is a collection of unique keys and their values. They are similar to arrays but array use integer as an index and hash use any object type. They are also called associative arrays, dictionaries or maps. If a hash is accessed with a key that does not exist, the method will return nil. The Syntax is

```
name = {"key1" => "value1", "key2" => "value2", "key3" => "value3"...}
                OR
name = {key1:  'value1', key2:  'value2', key3:  'value3'...}
```

Creating Ruby Hash

Ruby hash is created by writing key-value pair within {} curly braces. To fetch a hash value, write the required key within [] square bracket.

Example:

```
1.  color = {
2.     "Rose" => "red",
3.     "Lily" => "purple",
4.     "Marigold" => "yellow",
5.     "Jasmine" => "white"
6.   }
7.   puts color['Rose']
8.   puts color['Lily']
9.   puts color['Marigold']
10.  puts color['Jasmine']
```

Output:

Red
Purple
Yellow
white

Modifying Ruby Hash

A Ruby hash can be modified by adding or removing a key value pair in an already existing hash.

Example:

```
1.  color = {
2.     "Rose" => "red",
```

```
3.          "Lily" => "purple",
4.          "Marigold" => "yellow",
5.          "Jasmine" => "white"
6.        }
7.        color['Tulip'] = "pink"
8.        color.each do |key, value|
9.        puts "#{key} color is #{value}"
10.     end
```

Output:

Rose color is red
Lilly color is purple
Marigold color is yellow
Jasmine color is white
Tulip color is pink

Ruby Hash Methods

A Ruby hash has many methods. Some are public class methods and some public instance methods.

Public Class Methods

Method	Description
Hash[object]	Create a new hash with given objects.
new(obj)	Return a new empty hash.
try_convert(obj)	Try to convert obj into hash.

Public Instance Methods

Method	Description
hsh==other_hash	Two hashes are equal if they contain same key and value pair.
hsh[key]	Retrieve value from the respective key.
hsh[key] = value	Associates new value to the given key.
assoc(obj)	Compare obj in the hash.
clear	Remove all key value pair from hash.
compare_by_identity	Compare hash keys by their identity.

compare_by_identity?	Return true if hash compare its keys by their identity.
default(key=nil)	Return default value.
default = obj	Sets the default value.
delete(key)	Delete key value pair.
each	Call block once for each key in hash.
empty?	Return true if hash contains no key value pair.
eql>(other)	Return true if hash and other both have same content
fetch(key[, default])	Return value from hash for a given key.
flatten	Return a new array that is a one-dimensional flattening of this hash.
has_key?(key)	Return true if given key is present in hash.
has_value?(value)	Return true if given value is present in hash for a key.
include?(key)	Return true if given key is present in hash.
to_s/ inspect	Return content of hash as string.

Arrays

An array stores data or list of data. It can contain all types of data. Data in an array are separated by comma in between them and are enclosed by square bracket. For example,

```
["Akash", "Ankit", "Aman"]
```

Elements from an array are retrieved by their position. The position of elements in an array starts with 0.

Example:

```
1.  #!/usr/bin/ruby
2.  data = ["Akash", "Ankit", "Aman"]
3.  puts data[0]
4.  puts data[1]
5.  puts data[2]
```

Output:

```
Akash
Ankit
Aman
```

Ruby Arrays

Ruby arrays are ordered collections of objects. They can hold objects like integer, number, hash, string, symbol or any other array.

Its indexing starts with 0. The negative index starts with -1 from the end of the array. For example, -1 indicates last element of the array and 0 indicates first element of the array.

Creating Ruby Arrays

A Ruby array is created in many ways.

- ✓ Using literal constructor []
- ✓ Using new class method

Using literal construct []

A Ruby array is constructed using literal constructor []. A single array can contain different type of objects. For example, following array contains an integer, floating number and a string.

```
1.  exm = [4, 4.0, "Jose", ]
2.  puts exm
```

Output:

```
4
4.0
Jose
```

Using new class method

A Ruby array is constructed by calling ::new method with zero, one or more than one arguments. The syntax is

arrayName = Array.new

To set the size of an array,

Syntax is:

```
arrayName = Array.new(10)
```

Here, we have mentioned that array size is of 10 elements. To know the size of an array, either size or length method is used.

Example:

```
1.  #!/usr/bin/ruby
2.  exm = Array.new(10)
3.  puts exm.size
4.  puts exm.length
```

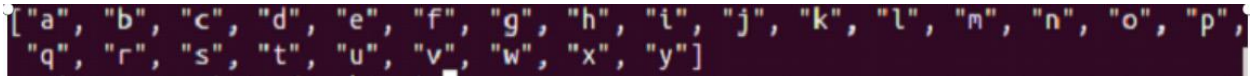
Output:

```
10
10
```

Example:

```
1.  #!/usr/bin/ruby
2.  exm = Array("a"..."z")
3.  puts "#{exm}"
```

Output:



Accessing Array Elements

Ruby array elements can be accessed using **#[]** method. You can pass one or more than one arguments or even a range of arguments. The general format is

#[] method

Example:

```
1.  days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2.  puts days[0]
3.  puts days[10]
4.  puts days[-2]
5.  puts days[2, 3]
6.  puts days[1..7]
```

Output:

```
Mon

Sat
Wed
Thu
Fri
Tue
Wed
Thu
Fri
Sat
Sun
```

at method : To access a particular element, at method can also be used.

Example:

```
1.  days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2.  puts days.at(0)
3.  puts days.at(-1)
4.  puts days.at(5)
```

Output:

```
Mon
Sun
Sat
```

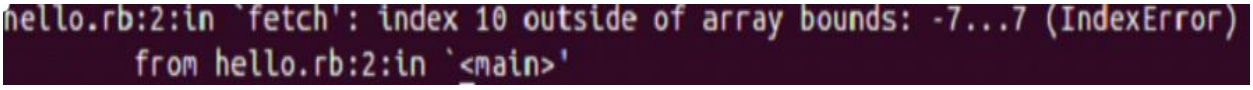
slice method : The slice method works similar to #[] method.

fetch method : The fetch method is used to provide a default value error for out of array range indices.

Example:

```
1.  days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2.  puts days.fetch(10)
```

Output:



Example:

```
1.  days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2.  puts days.fetch(10, "oops")
```

Output:

oops

first and last method : The first and last method will return first and last element of an array respectively.

Example:

```
1.  days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2.  puts days.first
3.  puts days.last
```

Output:

Mon
Sun

take method : The take method returns the first n elements of an array.

Example:

```
1.  days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2.  puts days.take(1)
3.  puts days.take(2)
```

Output:

Mon
Mon
Tue

drop method : The drop method is the opposite of take method. It returns elements after n elements have been dropped.

Example:

```
1.  days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2.  puts days.drop(5)
3.  puts days.drop(6)
```

Output:

Sat
Sun
Sun

Adding Items to Array : Ruby array elements can be added in different ways.

- ✓ push or <<
- ✓ unshift
- ✓ insert

push or << : Using push or <<, items can be added at the end of an array.

Example:

```
1.  days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2.  puts days.push("Today")
3.  puts days << ("Tomorrow")
```

Output:

```
Mon
Tue
Wed
Thu
Fri
Sat
Sun
Today
Mon
Tue
Wed
Thu
Fri
Sat
Sun
Today
Tomorrow
```

Unshift : Using unshift, a new element can be added at the beginning of an array.

Example:

```
1.  days = ["Fri", "Sat", "Sun"]
2.  puts days.unshift("Today")
```

Output:

```
Today
Fri
Sat
Sun
```

insert

Using insert, a new element can be added at any position in an array. Here, first we need to mention the index number at which we want to position the element.

Example:

```
1.  days = ["Fri", "Sat", "Sun"]
2.  puts days.insert(2, "Thursday")
```

Output:

```
Fri
Sat
Thursday
Sun
```

Removing Items from Array : Ruby array elements can be removed in different ways.

- ✓ pop
- ✓ shift
- ✓ delete
- ✓ uniq

Pop : Using pop, items can be removed from the end of an array. It returns the removed item.
Example:

```
1. days = ["Fri", "Sat", "Sun"]
2. puts days.pop
```

Output:
Sun

Shift : Using shift, items can be removed from the start of an array. It returns the removed item.
Example:

```
1.  days = ["Fri", "Sat", "Sun"]
2.  puts days.shift
```

Output:
Fri

Delete : Using delete, items can be removed from anywhere in an array. It returns the removed item.
Example:

```
1.  days = ["Fri", "Sat", "Sun"]
2.  puts days.delete("Sat")
```

Output:
Sat

Uniq : Using uniq, duplicate elements can be removed from an array. It returns the remaining array.
Example:

```
1.  days = ["Fri", "Sat", "Sun", "Sat"]
2.  puts days.uniq
```

Output:
Fri
Sat
Sun

Ruby Each Iterator

The Ruby each iterator returns all the elements from a hash or array.

Syntax:

```
(collection).each do |variable|
  code...
end
```

Here collection can be any array, range or hash.
Example:

```
1.  #!/usr/bin/ruby
2.  (1...5).each do |i|
```

```
3.          puts i
4.      end
```
Output:
1
2
3
4

Ruby Times Iterator

A loop is executed specified number of times by the times iterator. Loop will start from zero till one less than specified number.

Syntax:

```
x.times do |variable|
  code...
end
```

Here, at place of x we need to define number to iterate the loop.

Example:

```
1.      #!/usr/bin/ruby
2.      5.times do |n|
3.        puts n
4.      end
```
Output:
0
1
2
3
4

Ruby Upto and Downto Iterators

An upto iterator iterates from number x to number y.

Syntax:

```
x.upto(y) do |variable|
  code
end
```

Example:

```
1.      #!/usr/bin/ruby
2.      1.upto(5) do |n|
3.        puts n
4.      end
```
Output:
1
2
3
4
5

Ruby Step Iterator

A step iterator is used to iterate while skipping over a range.

Syntax:

BEGIN	do	next	then	break	false	rescue	when

```
(controller).step(x) do |variable|
  code
end
```

Here, x is the range which will be skipped during iteration.

Example:

```
1.  #!/usr/bin/ruby
2.  (10..50).step(5) do |n|
3.    puts n
4.  end
```

Output:

```
10
15
20
25
30
35
40
45
50
```

Ruby Each_Line Iterator

A each_line iterator is used to iterate over a new line in a string.

Example:

```
1.  #!/usr/bin/ruby
2.  "All\nthe\nwords\nare\nprinted\nin\na\nnew\line.".each_line do |line|
3.  puts line
4.  end
```

Output:

```
All
the
words
are
printed
in
a
newline.
```

Reserved Words

The following list shows the reserved words in Ruby. These reserved words may not be used as constant or variable names. They can, however, be used as method names.

END	else	nil	true	case	for	retry	while
alias	elsif	not	undef	class	if	return	while
and	end	or	unless	def	in	self	__FILE__
begin	ensure	redo	until	defined?	module	super	__LINE__

Ruby Modules

- ✓ Ruby module is a collection of methods and constants.
- ✓ A module method may be instance method or module method.
- ✓ Instance methods are methods in a class when module is included.
- ✓ Module methods may be called without creating an encapsulating object while instance methods may not.
- ✓ They are similar to classes as they hold a collection of methods, class definitions, constants and other modules.
- ✓ They are defined like classes. Objects or subclasses can not be created using modules. There is no module hierarchy of inheritance.

Modules basically serve two purposes:

- ✓ They act as namespace. They prevent the name clashes.
- ✓ They allow the mixin facility to share functionality between classes.

Syntax:

```
module ModuleName
  statement1
  statement2
  .....
end
```

Module name should start with a capital letter.

Module Namespaces

While writing larger files, a lot of reusable codes are generated. These codes are organized into classes, which can be inserted into a file.

For example, if two persons have the same method name in different files. And both the files need to be included in a third file. Then it may create a problem as the method name in both included files is same.

Here, module mechanism comes into play. Modules define a namespace in which you can define your methods and constants without over riding by other methods and constants.

Example:

Suppose, in file1.rb, we have defined number of different type of library books like fiction, horror, etc. In file2.rb, we have defined the number of novels read and left to read including fiction novels. In file3.rb, we need to load both the files file1 and file2. Here we will use module mechanism.

file1.rb

```
1.  #!/usr/bin/ruby
2.
```

```
# Module defined in file1.rb file

module Library
  num_of_books = 300
  def Library.fiction(120)
  # ..
  end
  def Library.horror(180)
  # ..
  end
end
```

file2.rb

```
#!/usr/bin/ruby

# Module defined in file2.rb file

module Novel
  total = 123
  read = 25
  def Novel.fiction(left)
  # ...
  end
end
```

file3.rb

```
require "Library"
require "Novel"

x = Library.fiction(Library::num_of_books)
y = Novel.fiction(Novel::total)
```

A module method is called by preceding its name with the module's name and a period, and you reference a constant using the module name and two colons.

Module Mixins

Ruby doesn't support multiple inheritance. Modules eliminate the need of multiple inheritance using mixin in Ruby.

A module doesn't have instances because it is not a class. However, a module can be included within a class.

When you include a module within a class, the class will have access to the methods of the module.

Example:

```
module Name
  def bella
  end
```

```
4.      def ana
5.      end
6.    end
7.    module Job
8.      def editor
9.      end
10.     def writer
11.     end
12.   end
13.
14.   class Combo
15.   include Name
16.   include Job
17.     def f
18.     end
19.   end
20.
21.   final=Combo.new
22.   final.bella
23.   final.ana
24.   final.editor
25.   final.writer
26.   final.f
```

Here, module Name consists of methods bella and ana. Module Job consists of methods editor and writer. The class Combo includes both the modules due to which class Combo can access all the four methods. Hence, class Combo works as **mixin.**

The methods of a module that are mixed into a class can either be an instance method or a class method. It depends upon how you add mixin to the class.

5.1.2. Ruby Operators

Ruby has a built-in modern set of operators. Operators are a symbol which is used to perform different operations. For example, +, -, /, *, etc.

Types of operators:

- ✓ Unary operator
- ✓ Airthmetic operator
- ✓ Bitwise operator
- ✓ Logical operator
- ✓ Ternary operator
- ✓ Assignment operator
- ✓ Comparison operator
- ✓ Range operator

Unary Operator

Unary operators expect a single operand to run on.

! - Boolean NOT

~ - Bitwise complement

+ - Unary plus

Example

In file hello.rb, write the following code.

```
1. #!/usr/bin/ruby -w
2. puts("Unary operator")
3. puts(~5)
4. puts(~-5)
5. puts(!true)
6. puts(!false)
```

Output:

```
Unary Operator
-6
4
false
true
```

Airthmetic Operator

Airthmetic operators take numerical values as operands and return them in a single value.

+ - Adds values from both sides of the operator.
- - Subtract values from both sides of the operator.
/ - Divide left side operand with right side operand.
* - Multiply values from both sides of the operator.
** - Right side operand becomes the exponent of left side operand.
% - Divide left side operand with right side operand returning remainder.

Example

In file hello.rb, write the following code.

```
1.  #!/usr/bin/ruby -w
2.  puts("add operator")
3.   puts(10 + 20)
4.   puts("subtract operator")
5.   puts(35 - 15)
6.   puts("multiply operator")
7.   puts(4 * 8)
8.   puts("division operator")
9.   puts(25 / 5)
10.  puts("exponential operator")
11.  puts(5 ** 2)
12.  puts("modulo operator")
13.  puts(25 % 4)
```

Output:

```
add operator
30
subtract operator
20
multiply operator
32
division operator
```

5
exponential operator
25
modulo operator
1

Bitwise Operator

Bitwise operators work on bits operands.

& - AND operator
| - OR operator
<< - Left shift operator
>> - Right shift operator
^ - XOR operator
~ - Complement operator

Logical Operator

Logical operators work on bits operands.

&& - AND operator
|| - OR operator

Ternary Operator

Ternary operators first check whether given conditions are true or false, then execute the condition.

?: - Conditional expression

Example

In file hello.rb, write the following code.

```
1.  #!/usr/bin/ruby -w
2.  puts("Ternary operator")
3.  puts(2<5 ? 5:2)
4.  puts(5<2 ? 5:2)
```

Output:

Ternary operator
5
2

Assignment operator

Assignment operator assign a value to the operands.

= - Simple assignment operator
+= - Add assignment operator
-= - Subtraction assignment operator
*= - Multiply assignment operator
/= - Divide assignment operator

Comparison Operator

Comparison operators compare two operands.

== - Equal operator

!= - Not equal operator
> - left operand is greater than right operand
< - Right operand is greater than left operand
>= - Left operand is greater than or equal to right operand
<= - Right operand is greater than or equal to left operand
<=> - Combined comparison operator
.eql? - Checks for equality and type of the operands
equal? - Checks for the object ID

Example

In file hello.rb, write the following code.

```
1.  #!/usr/bin/ruby -w
2.   puts("Comparison operator")
3.   puts(2 == 5)
4.   puts(2 != 5)
5.   puts(2 > 5)
6.   puts(2 < 5)
7.   puts(2 >= 5)
8.   puts(2 <= 5)
```

Output:

Comparison operator
false
true
false
true
false
true

Range Operator

Range operators create a range of successive values consisting of a start, end and range of values in between. The (..) creates a range including the last term and (...) creates a range excluding the last term. For example, for the range of 1..5, output will range from 1 to 5. and for the range of 1...5, output will range from 1 to 4.

.. - Range is inclusive of the last term
… - Range is exclusive of the last term

5.2.Control Statements

Ruby If-else Statement

The Ruby if else statement is used to test condition. There are various types of if statement in Ruby.

- ✓ if statement
- ✓ if-else statement
- ✓ if-else-if (elsif) statement
- ✓ ternay (shortened if statement) statement

Ruby if statement

Ruby if statement tests the condition. The if block statement is executed if condition is true. **Syntax:**

```
if (condition)
//code to be executed
end
```

Example:

```
1.  a = gets.chomp.to_i
2.  if a >= 18
3.    puts "You are eligible to vote."
4.  end
```

Output:

Ruby if else

23
You are eligible to vote.

12Ruby if else statement tests the condition. The if block statement is executed if condition is true otherwise else block statement is executed. Syntax is:

```
if(condition)
   //code if condition is true
else
//code if condition is false
end
```

Example:

```
1.  a = gets.chomp.to_i
2.  if a >= 18
3.  puts "You are eligible to vote."
4.  else
5.  puts "You are not eligible to vote."
6.  end
```

Output:
15
You are not eligible to vote.
18
You are eligible to vote.
20
You are not eligible to vote.

Ruby if else if (elsif)

Ruby if else if statement tests the condition. The if block statement is executed if condition is true otherwise else block statement is executed. Syntax:

```
if(condition1)
//code to be executed if condition1is true
elsif (condition2)
//code to be executed if condition2 is true
else (condition3)
//code to be executed if condition3 is true
```

```
end
```

Example:

```
1.  a = gets.chomp.to_i
2.  if a <50
3.    puts "Student is fail"
4.  elsif a >= 50 && a <= 60
5.    puts "Student gets D grade"
6.  elsif a >= 70 && a <= 80
7.    puts "Student gets B grade"
8.  elsif a >= 80 && a <= 90
9.    puts "Student gets A grade"
10. elsif a >= 90 && a <= 100
11.   puts "Student gets A+ grade"
12. end
```

Output:

```
45
Student is fail
98
Student gets A+ grade
100
Student gets A+ grade
64
72
Student gets B grade
```

Ruby ternary Statement

In Ruby ternary statement, the if statement is shortened. First it evaluates an expression for true or false value then execute one of the statements. **Syntax:**

```
test-expression ? if-true-expression : if-false-expression
```

Example:

```
1.  var = gets.chomp.to_i;
2.  a = (var > 3 ? true : false);
3.  puts a
```

Output:

```
2
false
5
true
```

Ruby Case Statement

In Ruby, we use 'case' instead of 'switch' and 'when' instead of 'case'. The case statement matches one statement with multiple conditions just like a switch statement in other languages.

Syntax:

```
case expression
[when expression [, expression ...] [then]
  code ]...
```

```
[else
  code ]
end
```

Example:

```
1.  #!/usr/bin/ruby
2.  print "Enter your day: "
3.  day = gets.chomp
4.  case day
5.  when "Tuesday"
6.  puts 'Wear Red or Orange'
7.  when "Wednesday"
8.  puts 'Wear Green'
9.  when "Thursday"
10. puts 'Wear Yellow'
11. when "Friday"
12. puts 'Wear White'
13. when "Saturday"
14. puts 'Wear Black'
15. else
16. puts "Wear Any color"
17. end
```

Output:

```
Enter your day: Sunday
Wear Any color
Enter your day: Saturday
Wear Black
Enter your day: Saturday
Wear Any color
```

Look at the above output, conditions are case sensitive. Hence, the output for 'Saturday' and 'saturday' are different.

Ruby for Loop

Ruby for loop iterates over a specific range of numbers. Hence, for loop is used if a program has fixed number of iterations. Ruby for loop will execute once for each element in expression. The general Syntax is:

```
for variable [, variable ...] in expression [do]
  code
end
```

Ruby for loop using range

```
1.  a = gets.chomp.to_i
2.  for i in 1..a do
3.    puts i
4.  end
```

Output:
5
1
2
3
4
5

Ruby for loop using array

```
1.  x = ["Blue", "Red", "Green", "Yellow", "White"]
2.  for i in x do
3.    puts i
4.  end
```

Output:
Blue
Red
Green
Yellow
White

Ruby while Loop

The Ruby while loop is used to iterate a program several times. If the number of iterations is not fixed for a program, while loop is used. Ruby while loop executes a condition while a condition is true. Once the condition becomes false, while loop stops its execution. The general syntax is :

```
while conditional [do]
  code
end
```

Example

```
1.  #!/usr/bin/ruby
2.  x = gets.chomp.to_i
3.  while x >= 0
4.    puts x
5.    x -=1
6.  end
```

Output:
5
5
4
3
2
1
0

Ruby do while Loop

The Ruby do while loop iterates a part of program several times. It is quite similar to a while loop with the only difference that loop will execute at least once. It is due to the fact that in do while loop, condition is written at the end of the code. The general format is

```
loop do
  #code to be executed
  break if booleanExpression
end
```

Example:

```
1.  loop do
2.    puts "Checking for answer"
3.    answer = gets.chomp
4.    if answer != '5'
5.      break
6.    end
7.  end
```

Output:1

Checking for answer
3

Output:2

Checking for answer
5
Checking for answer
9

Ruby Until Loop

The Ruby until loop runs until the given condition evaluates to true. It exits the loop when condition becomes true. It is just opposite of the while loop which runs until the given condition evaluates to false. The until loop allows you to write code which is more readable and logical. The general Syntax is :

```
until conditional
  code
end
```

Example:

```
1.  i = 1
2.  until i == 5
3.     print i*10, "\n"
4.     i += 1
5.  end
```

Output:

10
20
30
40
50

Ruby Break Statement

The Ruby break statement is used to terminate a loop. It is mostly used in while loop where value is printed till the condition is true, then break statement terminates the loop. The break statement is called from inside the loop. Syntax is :

```
break
```

Example:

```
1.  i = 1
2.  while true
3.    if i*5 >= 25
4.      break
5.    end
6.    puts i*5
7.    i += 1
8.  end
```

Output:

```
5
10
15
20
```

Ruby Next Statement

The Ruby next statement is used to skip loop's next iteration. Once the next statement is executed, no further iteration will be performed. The next statement in Ruby is equivalent to continue statement in other languages. The general syntax is :

```
next
```

Example:

```
1.  for i in 5...11
2.    if i == 7 then
3.      next
4.    end
5.    puts i
6.  end
```

Output:

```
5
6
8
9
10
```

Ruby redo Statement

Ruby redo statement is used to repeat the current iteration of the loop. The redo statement is executed without evaluating the loop's condition. The redo statement is used inside a loop. The general syntax is :

```
redo
```

Example:

```
1.  i = 0
```

```
2.      while(i < 5)   # Prints "012345" instead of "01234"
3.        puts i
4.        i += 1
5.         redo if i == 5
6.      end
```

Output:

0
1
2
3
4
5

Ruby retry Statement

Ruby retry statement is used to repeat the whole loop iteration from the start. The retry statement is used inside a loop. The syntax is **:**

retry

Ruby Comments

Ruby comments are non executable lines in a program. These lines are ignored by the interpreter hence they don't execute while execution of a program. They are written by a programmer to explain their code so that others who look at the code will understand it in a better way.

Types of Ruby comments:

1. Single line comment
2. multi line comment

Ruby Single Line Comment

The Ruby single line comment is used to comment only one line at a time. They are defined with **#** character. The general format is

#This is single line comment.

Example:

```
1.      i = 10  #Here i is a variable.
2.      puts i
```

Output:

10

The Ruby multi line comment is used to comment multiple lines at a time. They are defined with **=begin** at the starting and **=end** at the end of the line. The Syntax is :

```
=begin
   This
   is
   multi line
   comment
=end
```

Example:

```
1.  =begin
2.  we are declaring
3.  a variable i
4.  in this program
5.  =end
6.  i = 10
7.  puts i
```

Output:
10

5.3. Ruby Blocks

Ruby code blocks are called closures in other programming languages. It consist of a group of codes which is always enclosed with braces or written between **do..end.** The braces syntax always have the higher precedence over the do..end syntax. Braces have high precedence and do has low precedence.

A block is written in two ways,

- ✓ Multi-line between do and end (multi-line blocks are niot inline)
- ✓ Inline between braces {}

Both are same and have the same functionality. To invoke a block, you need to have a function with the same name as the block. A block is always invoked with a function. Blocks can have their own arguments. The general syntax is :

```
block_name{
  statement1
  statement2
  .....
}
```

Example:

The below example shows the **multi-line** block.

```
1.  [10, 20, 30].each do |n|
2.   puts n
3.  end
```

Output:
10
20
30

Below example shows the **inline** block.

```
[10, 20, 30].each {|n| puts n}
```

Output:
10
20
30

Ampersand parameter (&block)

The &block is a way to pass a reference (instead of a local variable) to the block to a method. Here, block word after the **&** is just a name for the reference, any other name can be used instead of this.

Example:

```
1.  def met(&block)
2.    puts "This is method"
3.    block.call
4.  end
5.  met { puts "This is &block example" }
```

Output:

This is method
This is &block example

Here, the block variable inside method **met** is a reference to the block. It is executed with the call method. The call method is same as yield method.

Initializing objects with default values

Ruby has an initializer called **yield(self)**. Here, **self** is the object being initialized.

Example:

```
1.  class Novel
2.    attr_accessor :pages, :category
3.    def initialize
4.      yield(self)
5.    end
6.  end
7.   novel = Novel.new do |n|
8.   n.pages = 564
9.   n.category = "thriller"
10. end
11. puts "I am reading a #{novel.category} novel which has #{novel.pages} pages."
```

Output:

I am reading a #{novel.category} novel which has 564 pages.

Ruby BEGIN Statement

Declares *code* to be called before the program is run.

```
BEGIN
 {
        code
 }
```

Ruby END Statement

Declares *code* to be called at the end of the program.

```
END
```

```
{
        code
}
```

Example

```
#!/usr/bin/ruby

puts "This is main Ruby Program"

END {
  puts "Terminating Ruby Program"
}
BEGIN {
  puts "Initializing Ruby Program"
}
```

This will produce the following result −

```
Initializing Ruby Program
This is main Ruby Program
Terminating Ruby Program
```

5.4. Ruby Class and Object

In object-oriented programming language, we design programs using objects and classes. Object is a physical as well as logical entity whereas class is a logical entity only.

Ruby Class

Ruby class defines blueprint of a data type. Each Ruby class is an instance of class **Class**. Classes in Ruby are first-class objects. Ruby class always starts with the keyword **class** followed by the class name. Conventionally, for class name we use CamelCase. The class name should always start with a capital letter. Defining class is finished with **end** keyword. Class name with more than one word run together with each word capitalized and no separating characters.

The general syntax is

```
class ClassName
   codes...
end
```

For example

```
class Java
    def initialize(name = “world”)
        @name = name
    end
    def say_welcome()
        puts “Welcome  #{@name}!”
    end
    def say_end()
        puts “Bye  #{@name}, see you again”
    end
End
```

A new class **Java** is created. The @name is an instance variable available to all the methods of the Java class. It is used by say_welcome and say_bye.

Ruby Object

In Ruby, everything is an object. When we create objects, they communicate together through methods. Hence, an object is a combination of data and methods.

To create an object, first, we define a class. Single class can be used to create many objects. Objects are declared using **new** keyword.

Creating object

Objects in Ruby are created by calling **new** method of the class. It is a unique type of method and predefined in the Ruby library. Ruby objects are instances of the class. The general syntax is

objectName = className.new

Example:

We have a class named **Java**. Now, let's create an object **java** and use it with following command,

```
java = Java.new("John")
java.say_welcome
java.say_bye
```

Assigning @name =John, and it will produce

```
Welcome John!
Bye John, see you again
```

Ruby Methods

Ruby methods prevent us from writing the same code in a program again and again. It is a set of expression that returns a value. Ruby methods are similar to the functions in other languages. They unite one or more repeatable statements into one single bundle.

Defining Method

Methods are functions which are defined inside the body of a class. Data in Ruby is accessible only via methods. There is a follow path in which Ruby looks when a method is called. To find out the method lookup chain we can use **ancestors** method.

To use a method, we need to first define it. Ruby method is defined with the **def** keyword followed by method name. At the end we need to use **end** keyword to denote that method has been defined.

Methods name should always start with a lowercase letter. Otherwise, it may be misunderstood as a constant. **Syntax is :**

```
def methodName
   code...
end
```

Example:

Here, we have defined a method **say_welcome** using def keyword. The last line end keyword says that we are done with the method defining. Now let's call this method. A method is called by just writing its name.

```
>say_welcome
```

This will produce the following output

welcome john!

Defining Method with Parameter

To call a particular person, we can define a method with parameter.

```
def welcome(name)
        puts “Hello #{name}, welcome at Ruby class”
end
```

Here, #{name} is a way in Ruby to insert something into string. The bit inside the braces is turned into a string. Let's call the method by passing a parameter **Edward.**

welcome(“Karthi”)

Output is

Hello Karthi, welcome at Ruby class

Instance Methods

The instance methods are also defined with def keyword and they can be used using a class instance only.

Example:

```
1.  #!/usr/bin/ruby -w
2.
3.  # define a class
4.  class Circle
5.    # constructor method
6.    def initialize(r)
7.      @radius = r
8.    end
9.    # instance method
10.   def getArea
11.     3.14 * @radius * @radius
12.   end
13. end
14.
15. # create an object
16. circle = Circle.new(2)
17.
18. # call instance methods
19. a = circle.getArea()
20. puts "Area of the box is : #{a}"
```

Output:

Area of the box is : 12.56

Ruby OOPs Concept

Ruby is a true object oriented language which can be embedded into Hypertext Markup Language. Everything in Ruby is an object. All the numbers, strings or even class is an object. The whole Ruby language is basically built on the concepts of object and data.

OOPs is a programming concept that uses objects and their interactions to design applications and computer programs. Following are some basic concepts in OOPs:

- ✓ Encapsulation
- ✓ Polymorphism
- ✓ Inheritance
- ✓ Abstraction

Encapsulation: It hides the implementation details of a class from other objects due to which a class is unavailable to the rest of the code. Its main purpose is to protect data from data manipulation.

Polymorphism: It is the ability to represent an operator or function in different ways for different data input.

Inheritance: It creates new classes from pre defined classes. New class inherit behaviors of its parent class which is referred as superclass. In this way, pre defined classes can be made more reusable and useful.

Abstraction: It hides the complexity of a class by modelling classes appropriate to the problem.

Encapsulation:

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. In a different way, encapsulation is a protective shield that prevents the data from being accessed by the code outside this shield.

- ✓ Technically in encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of own class in which they are declared.
- ✓ Encapsulation can be achieved by declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

Example:

```
# Ruby program to illustrate encapsulation
#!/usr/bin/ruby
class Demoencapsulation
    def initialize(id, name, addr)
        # Instance Variables
        @cust_id = id
        @cust_name = name
        @cust_addr = addr
     end
 # displaying result
    def display_details()
         puts "Customer id: #@cust_id"
         puts "Customer name: #@cust_name"
         puts "Customer address: #@cust_addr"
    end
end
# Create Objects
cust1 = Demoencapsulation .new("1", "Mike", "Wisdom Apartments, Ludhiya")
cust2 = Demoencapsulation .new("2", "Jackey", "New Empire road, Khandala")
```

```
# Call Methods
cust1.display_details()
cust2.display_details()
```

Output:

```
Customer id: 1

Customer name: Mike

Customer address: Wisdom Apartments, Ludhiya

Customer id: 2

Customer name: Jackey

Customer address: New Empire road, Khandala
```

Explanation: In the above program, the class Demoencapsulation encapsulate the methods of the class. You can only access these methods with the help of objects of the Demoencapsulation class i.e. cust1 and cust2.

Advantages of Encapsulation:

- ✓ **Data Hiding:**The user will have no idea about the inner implementation of the class. It will not be visible to the user that how the class is storing values in the variables. He only knows that we are passing the values to a setter method and variables are getting initialized with that value.
- ✓ **Reusability:** Encapsulation also improves the re-usability and easy to change with new requirements.
- ✓ **Testing code is easy:**Encapsulated code is easy to test for unit testing.

Ruby Inheritance

In inheritance, we create new classes using pre defined classes. Newly created classes are called derived classes and classes from which they are derived are called base classes. With inheritance, a code can be reused again which reduces the complexity of a program. Ruby does not support multiple levels of inheritance. Instead it supports **mixins.** In Ruby, < character is used to create a subclass. The syntax is shown below:

parentClass < subClass

Example:

```
1.  #!/usr/bin/ruby
2.
3.  class Parent
4.
5.     def initialize
6.        puts "Parent class created"
7.     end
8.  end
9.
10. class Child < Parent
11.
12.    def initialize
```

```
13.         super
14.         puts "Child class created"
15.       end
16.     end
17.     Parent.new
18.     Child.new
```

In the above example, two classes are created. One is base **Parent** class and other is derived **Child** class. The **super** method calls the constructor of the Parent class.From the last two line, we instantiate both the classes.

Output:

```
Parent class created
Parent class created
Child class created
```

In the output, first the Parent class is created, derived Child class also calls the constructor of its parent class and then Child class is created.

Ruby Constructor

A constructor is automatically called when an object is created. They do not return any values. In Ruby, they are called **initialize.** A constructor's main purpose is to initiate the state of an object. They can't be inherited. The parent object constructor is called with super method.

Example:

```
1.      #!/usr/bin/ruby
2.       class Parent
3.               def initialize
4.                       puts "Parent is created"
5.               end
6.        end
7.        Parent.new
```

Output:

```
Parent is created
```

Polymorphism

In Ruby, one does not have anything like the variable types as there is in other programming languages. Every variable is an “object” which can be individually modified. One can easily add methods and functions on every object. So here, the Object Oriented Programming plays a major role. There are many pillars of Object Oriented Programming in every other programming language, like Inheritance, Encapsulation etc. One of these pillars is Polymorphism.

Polymorphism is a made up of two words **Poly** which means Many and **Morph** which means Forms. So Polymorphism is a method where one is able to execute the same method using different objects. In polymorphism, we can obtain different results using the same function by passing different input objects. One can also write the If-Else commands but that just makes the code more lengthy. To avoid this, the programmers came up with the concept of polymorphism.

In Polymorphism, classes have different functionality but they share common interference. The concept of polymorphism can be studied under few sub categories.

1. Polymorphism using Inheritance
2. Polymorphism using Duck-Typing

1. **Polymorphism using inheritance**

Inheritance is a property where a child class inherits the properties and methods of a parent class. One can easily implement polymorphism using inheritance. It can be explained using the following example:

```
# Ruby program of Polymorphism using inheritance
class Vehicle
   def tyreType
      puts "Heavy Car"
   end
end

# Using inheritance
class Car < Vehicle
   def tyreType
      puts "Small Car"
   end
end

# Using inheritance
class Truck < Vehicle
   def tyreType
      puts "Big Car"
   end
end

# Creating object
vehicle = Vehicle.new
vehicle.tyreType()

# Creating different object calling same function
vehicle = Car.new
vehicle.tyreType()

# Creating different object calling same function
vehicle = Truck.new
vehicle.tyreType()
```

Output:

```
Heavy Car

Small Car

Big Car
```

The above code is a very simple way of executing basic polymorphism. Here, the **tyreType** method is called using different objects like Car and Truck. The Car and Truck classes both are the child classes of Vehicle. They both inherit the methods of vehicle class (primarily the tyretype method).

Polymorphism using Duck-Typing

In Ruby, we focus on the object's capabilities and features rather than its class. So, Duck Typing is nothing but working on the idea of what an object can do rather than what it actually is. Or, what operations could be performed on the object rather than the class of the object. Here is a small program to represent the before mentioned process.

Example :

```
# Ruby program of polymorphism using Duck typing
# Creating three different classes
class Hotel
      def enters
           puts "A customer enters"
      end
      def type(customer)
          customer.type
      end
      def room(customer)
          customer.room
      end
end
 # Creating class with two methods
class Single
      def type
           puts "Room is on the fourth floor."
      end
      def room
          puts "Per night stay is 5 thousand"
      end
end
class Couple
       # Same methods as in class single
       def type
            puts "Room is on the second floor"
       end
       def room
            puts "Per night stay is 8 thousand"
        end
end
# Creating Object
# Performing polymorphism
hotel= Hotel.new
```

```
puts "This visitor is Single."
customer = Single.new
hotel.type(customer)
hotel.room(customer)
puts "The visitors are a couple."
customer = Couple.new
hotel.type(customer)
hotel.room(customer)
```

Output :

```
This visitor is Single.

Room is on the fourth floor.

Per night stay is 5 thousand

The visitors are a couple.

Room is on the second floor

Per night stay is 8 thousand
```

In the above example, The customer object plays a role in working with the properties of the customer such as its “type” and its “room”. This is an example of polymorphism.

Data Abstraction in Ruby

The idea of representing significant details and hiding details of functionality is called data abstraction. The interface and the implementation are isolated by this programming technique. Data abstraction is one of the object oriented programming features as well. Abstraction is trying to minimize information so that the developer can concentrate on a few ideas at a time. Abstraction is the foundation for software development.

Consider a real-life example of making a phone call. The only thing the person knows is that typing the numbers and hitting the dial button will make a phone call, they don’t know about the inner system of the phone or the dial button on the phone. That’s what we call abstraction.
Another real-life example of abstraction is as users of television sets, we can switch it on or off, change the channel and set the volume without knowing the details about how its functionality has been implemented.

Data Abstraction in modules:

In Ruby, Modules are defined as a set of methods, classes, and constants together.For example, consider the sqrt() method present in Math module. Whenever we need to calculate the square root of a non negative number, We simply call the sqrt() method present in the Math module and send the number as a parameter without understanding the actual algorithm that actually calculates the square root of the numbers.

Data Abstraction in Classes: we can use classes to perform data abstraction in ruby. The class allows us to group information and methods using access specifiers (private, protected, public). The Class will determine which information should be visible and which is not.

Data Abstraction using Access Control: There are three levels of access control in Ruby (private, protected, public). These are the most important implementation of data abstraction in ruby.

For Example

- ✓ Members who have been declared public in a class can be accessed from anywhere in the program.
- ✓ Members declared to be private in a class can only be accessed from within the class. They are not allowed to access any part of the code outside the class.

Output:

```
In Public
In Private
```

In the above program, we are not allowed to access the privateMethod(of Geeks class directly, however, we can call the publicMethod() in the class in order to access the privateMethod(.

```
# Ruby program to demonstrate Data Abstraction
class Geeks
# defining publicMethod
public
    def publicMethod
        Puts "In Public!"
        # calling privateMethod inside publicMethod
        privateMethod
    end
    # defining privateMethod
private
    def privateMethod
        puts "In Private!"
    end
end
 # creating an object of class Geeks
obj = Geeks.new
 # calling the public method of class Geeks
obj.publicMethod
```

Advantages of Data Abstraction:

- ✓ Helps increase the security of a system because only crucial details are made available to the user.
- ✓ It increases re-usability and prevents redundancy of code.
- ✓ Could alter the internal class implementation independently without affecting the user.

5.5. Welcome to rails

5.5. Ruby on Rails

Ruby on Rails tutorial provides basic and advanced concepts of Ruby on Rails.

5.5.1.Ruby on Rails Introduction

Ruby on Rails is a server-side web application development framework written in Ruby language by David Heinemeier Hansson. He was working at 37 signals (now Basecamp) company to create a project management application in Ruby. To help speed along the process, he created a custom web framework Ruby on Rails. It is also called Rails.

It allows you to write less code than other languages and frameworks. It includes everything needed to create database-backed web applications according to MVC pattern.

Ruby on Rails Installation

We will set up Ruby on Rails in Ubuntu 14.04 operating system.There are three methods to install Ruby:

- ✓ Using rbenv (recommended)
- ✓ Using rvm
- ✓ From source

We will install using rbenv as it is the most recommended way. First we will install some dependencies for Ruby:

sudo apt-get update

1. sudo apt-get install git-core curl zlib1g-dev build-essential libssl-dev
2. libreadline-dev libyaml-dev libsqlite3-dev sqlite3 libxml2-dev
3. libxslt1-dev libcurl4-openssl-dev python-software-properties libffi-dev nodejs

Install rbenv

Installing rbenv is a simple two way process. First rbenv will be installed and then ruby-build. Follow the following commands:

1. cd
2. git clone git://github.com/sstephenson/rbenv.git .rbenv
3. echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc\
4. echo 'eval "$(rbenv init -)"' >> ~/.bashrc
5. exec $SHELL

1. git clone git://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
2. echo 'export PATH="$HOME/.rbenv/plugins/ruby-build/bin:$PATH"' >> ~/.bashrc
3. exec $SHELL

The above command will install rbenv in your home directory and will set the appropriate environment variables.

Install Ruby

Install Ruby using following commands:

1. rbenv install -v 2.2.3
2. rbenv global 2.2.3

To disable Rubygems which generate local documentation for each gem that you install, use following command:

1. echo "gem: --no-document" > ~/.gemrc

Now you need to install bundler gem to manage application dependencies with following command.

1. gem install bundler

Install Rails

Install Rails using following command,

```
1. gem install rails
```

You can specify the version of Rails which you want to install using -v option in the above command.

Now we will run **rehash** sub-command. This will install shims for all Ruby executables known to rbenv, which allow you to use executables.

```
1. rbenv rehash
```

To verify the installed Rails version, use the following command.

```
1. rails -v
```

Install JavaScript Runtime

Some Rails features like Asset Pipeline, depends on JavaScript runtime. To get this functionality, we need to install Node.js.

```
1. sudo add-apt-repository ppa:chris-lea/node.js
```

Now update apt-get and install Node.js packet.

```
1. sudo apt-get update
2. sudo apt-get install nodejs
```

Now you have successfully installed Ruby on Rails on your system.

Install Database

Rails default database is Sqlite3. If you want to use some other database due to any reason, then you need to install it. Here, we will install MySQL server as our database.

```
1. sudo apt-get install mysql-server mysql-client libmysqlclient-dev
```

After this, install mysql2 gem, with following command.

```
1. gem install mysql2
```

Now you can easily use MySQL with Rails in your system.

Rails IDE or Editor

Ruby on Rails can be used with either a simple text editor or with an IDE. A text editor is a tool that creates and edits a file with only plain text. Once the code is written in the editor, it need to be compiled and run on a command line tool.

An IDE stands for Integrated Development Environment. It is a more powerful tool providing many features, including text editor features. Some of the Rails IDEs are listed below:

TextMate , E, Intellij IDEA, NetBeans, Eclipse, Heroku, Aptana Studio, RubyMine, Kuso, IDE, Komodo, Redcar, Arcadia, Ice Coder etc…

Rails Scripts

Rails provides us some excellent tools that are used to develop Rails application. These tools are packaged as scripts from command line. Following are the most useful Rails scripts used in Rails application:

- ✓ Rails Console
- ✓ WEBrick Web Server

- ✓ Generators
- ✓ Migrations

Ruby on Rails 5 Hello World Example

Step 1 : Create a directory **jtp** in which all the code will be present and will navigate from the command line.

mkdir jtp

Step 2 : Change the directory to jtp

cd jtp

Step 3 : Create a new application with the name **helloWorld.**

rails new helloWorld

You will see something as shown in the below snapshot.

```
create
create  README.md
create  Rakefile
create  config.ru
create  .gitignore
create  Gemfile
create  app
create  app/assets/config/manifest.js
create  app/assets/javascripts/application.js
create  app/assets/javascripts/cable.js
```

A helloWorld directory will be created in your system. Inside this folder there will be many files and subfolders which is actually the Rails application.

Step 4 : Move in to your above created application directory that is helloWorld.

cd helloWorld

Step 5 : Rails 5 has no longer a static index page in production. There will not be a root page in the production, so we need to create it. First we will create a controller called **hello** for our home page.

1. **rails generate controller hello**

You will see something as shown in the below snapshot.

```
:~/jtp/helloWorld$ rails generate controller hello
```

```
Running via Spring preloader in process 7499
      create  app/controllers/hello_controller.rb
      invoke  erb
      create    app/views/hello
      invoke  test_unit
      create    test/controllers/hello_controller_test.rb
      invoke  helper
      create    app/helpers/hello_helper.rb
      invoke    test_unit
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/hello.coffee
      invoke    scss
```

Step 6 : Now we need to add an index page. In file app/views/hello/index.html.erb, write

1. <h2>Hello World</h2>
2.
3. Today is 23r March, Thursday.
4.

Step 7 : Now we need to route the Rails to this action. Edit the config/routes.rb file to set the index page to our new method. Add the following line in the routes.rb file,

root 'hello#index'

Step 8 : Now you can verify the page by running your server.

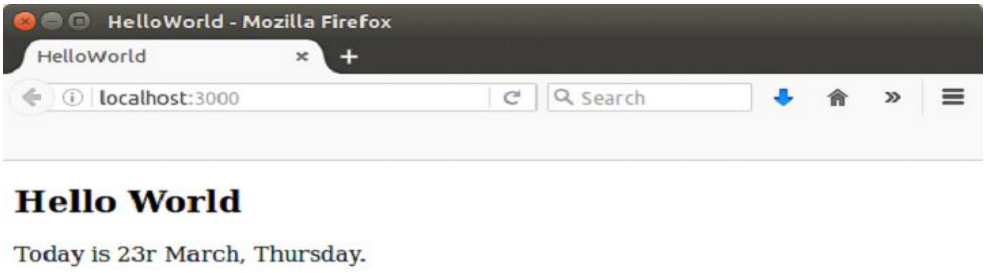
rails server

```
$  rails server
=> Booting Puma
=> Rails 5.0.2 application starting in development on http://localhost:3000
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.8.2 (ruby 2.2.3-p173), codename: Sassy Salamander
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
```

By default, Rails server listens to the port 3000. Although you can change it with the following command.

rails server -p portNumber

Step 9 :Visit click here in your browser.



5.5.2. Creating the Application Framework

Rails can do most of the work in creating your application. In fact, all you need to do is use the command rails applicationName, rails first at the command line in the rubydev directory. Rails creates the files you're going to need:

```
C:\rubydev\ch04>rails first
create
create app/controllers
create app/helpers
create app/models
create app/views/layouts
create config/environments
create components
create db
create doc
create lib
create lib/tasks
create log
create public/images
create public/javascripts
create public/stylesheets
create script/performance
create script/process
create test/fixtures
create test/functional
create test/integration
create test/mocks/development
create test/mocks/test
create test/unit
create vendor
create vendor/plugins
create tmp/sessions
create tmp/sockets
create tmp/cache
create Rakefile
create README
create app/controllers/application.rb
```

```
create app/helpers/application_helper.rb
create test/test_helper.rb
create config/database.yml
create config/routes.rb
create public/.htaccess
create config/boot.rb
create config/environment.rb
create config/environments/production.rb
create config/environments/development.rb
create config/environments/test.rb
create script/about
.
.etc…..
```

A lot of new directories (and only about half of them are shown here). Here is the new directory structure of the rubydev\msc\first directory:

```
rubydev
|__msc
    |__first
        |__README
        |__app
        |   |__controllers
        |   |__models
        |   |__views
        |   |__helpers
        |
        |__config
        |__components
        |__db
        |__doc
        |__lib
        |__public
        |__script
        |__test
        |__tmp
        |__vendor
```

The README document that is automatically generated and placed in the rubydev\msc\first directory contains an explanation of these directories.

app : Holds all the code that's specific to this particular application.
app/controllers : Holds controllers that should be named like weblog_controller.rb for automated URL mapping. All controllers should descend from ActionController::Base.
app/models : Holds models that should be named like post.rb. Most models will descend from ActiveRecord::Base.

app/views : Holds the template files for the view that should be named like weblog/index.rhtml for the WeblogController#index action. All views use eRuby syntax. This directory can also be used to keep stylesheets, images, and so on that can be symlinked to public.
app/helpers : Holds view helpers that should be named like weblog_helper.rb.
app/apis : Holds API classes for web services.
config : Configuration files for the Rails environment, the routing map, the database, and other dependencies.
components : Self-contained mini-applications that can bundle together controllers, models, and views.
db : Contains the database schema in schema.rb. db/migrate contains all the sequence of Migrations for your schema.
lib : Application specific libraries. Basically, any kind of custom code that doesn't belong under controllers, models, or helpers. This directory is in the load path.
Public : The directory available for the web server. Contains subdirectories for images, stylesheets, and javascripts. Also contains the dispatchers and the default HTML files.
Script : Helper scripts for automation and generation.
Test : Unit and functional tests along with fixtures.
Vendor : External libraries that the application depends on. Also includes the plugins subdirectory. : This directory is in the load path.

5.5.3.Running the Application

To launch your new application, start by changing directories to the new first directory:

C:\rubydev\msc>cd first

There are a number of short Ruby programs in the rubydev\msc\first\script directory for use with your new application. The server script launches the web server WEBrick, which comes with Rails, so enter **ruby script/server** at the command line:

```
C:\rubydev\ch04\first>ruby script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with — help for options
[2006-05-16 12:26:22] INFO WEBrick 1.3.1
[2006-05-16 12:26:22] INFO ruby 1.8.2 (2004-12-25) [i386-mswin32]
[2006-05-16 12:26:22] INFO WEBrick::HTTPServer#start: pid=1684 port=3000
```

This starts the WEBrick server on port 3000 of your local host, which means you can access your application at the URL http://localhost:3000/. To see that, open a browser on your machine and navigate to that URL, as shown in Figure

The web page displays a cheery message, indicating that you're now riding the Rails. Not bad. To end the WEBrick session now, follow the directions displayed when WEBrick started, such as pressing Ctrl+C in Windows.

5.5.4.Creating the Controller

The process of making the application do something for you begins when you create a *controller* for the application. The controller is like the boss of the application: it's the overseer that makes things happen. Rails uses a model-view controller (MVC) architecture in its web applications. The controller part is essential to any application, so you're going to need to create one. After stopping WEBrick, in the rubydev\msc\first directory, use the Ruby command ruby script/generate controller Hello to create a new controller named Hello:

```
C:\rubydev\msc\first>ruby script/generate controller Hello
      exists app/controllers/
      exists app/helpers/
      create app/views/hello
      exists test/functional/
      create app/controllers/hello_controller.rb
      create test/functional/hello_controller_test.rb
      create app/helpers/hello_helper.rb
```

That creates a new controller for your application. The code for the controllers in your application appears in the rubydev\msc\first\controllers directory, and now you'll find a file named hello_controller.rb in that directory—that's the support file for your new controller.
This file is long and complex. Just kidding :

```
class HelloController < ApplicationController
end
```

Here's one of the places where Rails favors convention over configuration—your entire HelloController class inherits just about all it needs from the ApplicationController class. That class is supported in application.rb, also in the rubydev\ch04\first\controllers directory—here are its contents:

```
# Filters added to this controller will be run for all controllers in the application.
# Likewise, all the methods added will be available for all controllers.
class ApplicationController < ActionController::Base
end
```

In other words, ApplicationController inherits from ActionController::Base—that is, the Base class in the ActionController module.

Using the Rails Documentation

If you want, you can take a look at the Rails documentation for classes like ActionController::Base. To do so, just enter the command **gem_server** on the command line:

```
C:\rubydev\msc\first>gem_server
```

Then navigate your browser to http://localhost:8808 to see the Rails documentation.

Testing the Controller

How far can you get with a web application that has a controller? You can test that out immediately in your browser.

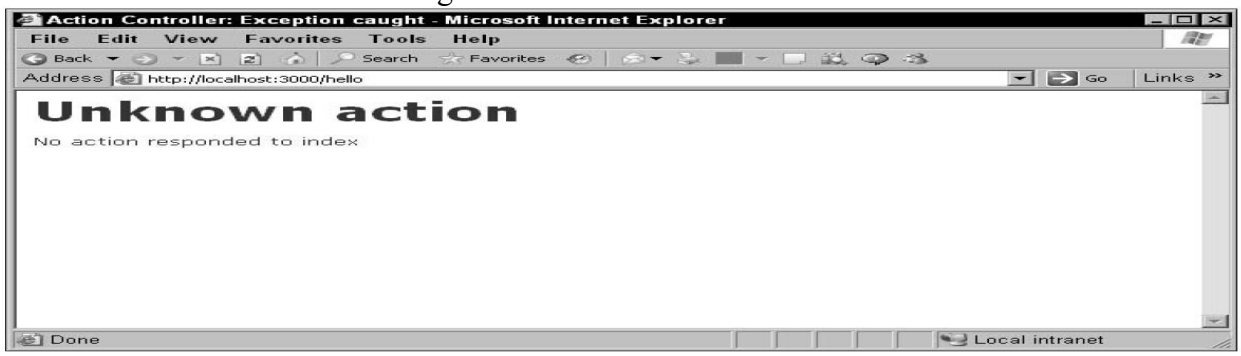
Example : Display a Message

To test a web application that has only a controller, follow these steps:

1. Start the WEBrick server:

C:\rubydev\msc\first>ruby script/server

2. Navigate to http://localhost:3000/hello.
3. You should see the results in Figure



Creating an Action

Controllers can execute actions to make a web application do something, and you can easily add actions to the web application in Rails. The idea is that a controller is the boss of the application, and it calls various actions, each of which performs a separate task. So you can think of web applications as collections of tasks, implemented by the actions, which are driven by the controller.

This example is going to have an action named there that will make the application display some text in your web browser. You can specify the action you want the controller to take in the URL you navigate to. To ask the hello controller to execute the there action, navigate to http://localhost:3000/hello/there—you just specify the controller's name first in the URL, followed by the action you want the controller to execute. Rails decodes the URL and sends your request to the appropriate controller, which in turn calls the appropriate action.

Creating the there action is easy in Rails—actions are supported by methods in the controller's .rb file. That's all there is to it.

Example : To add the action named there to the controller, follow these steps:

1. Edit your hello_controller.rb file in rubydev\ch04\first\app\controllers from this:

```
class HelloController < ApplicationController
end
```

to this, adding a new method named there:

```
class HelloController < ApplicationController
    def there
    end
end
```

2. Save hello_controller.rb.
3. Start the WEBrick server:

C:\rubydev\msc\first>ruby script/server

Navigate to http://localhost:3000/hello/there.

You should see the results shown as

Template is Missing

"Missing template ./script/../config/../app/views/hello/there.rhtml".

Creating a View

You've created a web application, added a controller to handle requests from the user, and added an action to let the controller respond to those requests. But you still need some way of returning a result to the user.

Associating a response with your action is done by creating a *view* in Rails applications. A view is just what it sounds like—a way of seeing some result. After your application is all done, it displays its results in a view. Ruby on Rails terminology—the controller is the boss of a web application, actions are tasks that controllers can perform, and views give the controller a way to display the results of the application.

You use a *template* to create a view. A template is a skeleton web page that will display your results in a browser; at runtime, an action can store data throughout that template so that your web page shows the data formatted as you want it.

Creating a template is easy—templates are just web pages with the extension .rhtml. That extension makes Rails read the file and pops into it any data from the action you want displayed before sending the template back to the browser.

In this example, the action is just a rudimentary, empty method named there in the controller:

```
class HelloController < ApplicationController
    def there
    end
end
```

The file, hello_controller.rb, is in rubydev\msc\first\app\controllers because it's the support code for the hello controller's there action. Rails will automatically connect a view template to this action if you give the template the action's name—there.rhtml, in this case—and place it in rubydev\msc\first\app\views\hello.

In other words, to establish a view template for the hello controller's there action, you can create a file named there.rhtml and store it in the rubydev\msc\first\app\views\hello directory.

Create a View

To add a view to the application, follow these steps:

1. Start your text editor and place this text in it:

```
<html>
    <head>
        <title>Using Ruby on Rails</title>
    </head>
    <body>
        <h1>Welcome to Ruby on Rails</h1>
        This is your first Ruby on Rails application.
        <br>
        <br>
        Using this application, you've been introduced to controllers, actions, and views.
        <br>
```

```
            <br>
                Not bad for a first example!
        </body>
</html>
```

2. Save this file as rubydev\msc\first\app\views\hello\there.rhtml.
3. Start the WEBrick server:

```
C:\rubydev\msc\first>ruby script/server
```

4.Navigate to http://localhost:3000/hello/there.

How It Works

This example shows how to connect a sample view template to an action, displaying a sample, static web page—there.rhtml—in the browser. All you had to do was to place there.rhtml into rubydev\msc\first\app\views\hello to connect the template to the hello controller's there action.

You've completed your first Ruby on Rails web application. Note that only two files were involved—hello_controller.rb and there.rhtml:

```
rubydev
    |__msc
        |__first
            |__README
            |__app
            |   |__controllers
            |       |__hello_controller.rb
            |   |__models
            |   |__views
            |       |__hello
            |           |__there.rhtml
            |   |__helpers
```

.

5.5.5. Introducing Model-View-Controller Architecture

Control starts in the browser, when the user enters the URL for the application. That sends a request to the web server, which decodes the URL and sends the request from the browser on to the controller. The controller can have a number of actions to select from—there's just one in this case, but you can add as many methods as you want to hello_controller.rb. The controller passes the request on to the appropriate action, as specified in the URL in this case.

The action in this example didn't really do anything, just used a view template to return data to the browser. The following figure shows what the process looks like schematically.

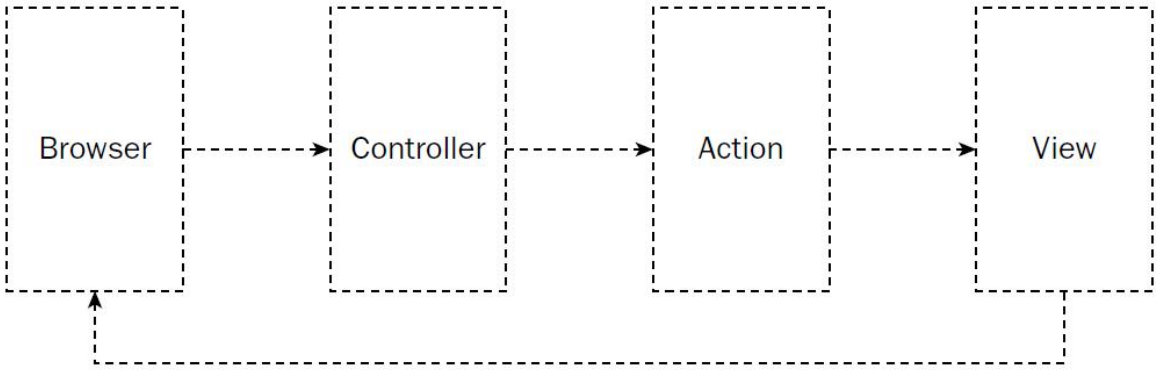


Figure 1 : This is a specific case of a more general picture—the model-view-controller picture.
Figure 2 : shows what that picture looks like in overview.

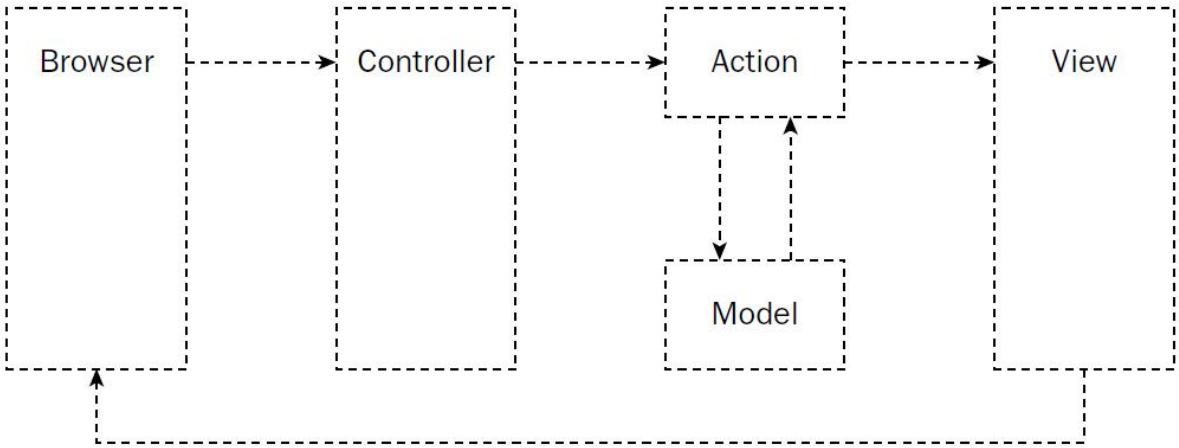


Figure 2:

It's going to be helpful to take this picture apart, piece by piece, to get the full model-view-controller story. In the early days of web applications, all the code was heaped together into a single document, which ran when you accessed it from a browser. However, as online applications got longer and longer, it became a good idea to split out the presentation code from the rest of the code to make maintenance and debugging easier. That led to a whole new breed of applications. But even that architecture in turn has been superceded by the addition of the model for data handling. Enter the MVC architecture, which all starts with the controller.

The Controller

When the user enters a URL into his browser, a request is sent from the browser to the web server. If your web server supports Rails, it's going to hand the request from the browser off to a Rails controller of the kind you've already seen.

The controller supervises the entire application, handling requests as needed. For example, the controller can decode the URL you've already seen—http://localhost:3000/hello/there—to know that you're requesting the action named there.

Controllers can also route requests between the web pages of an application—so far, the application you've seen only has one web page, but it's a rare application that stops at one web page. Most have multiple pages, and the controller can route the user from page to page.

At its most basic, a controller just inherits from the ApplicationController class, which in turn inherits from the ActionController::Base class—the ActionController module is the one that contains the support for controllers in Rails:

```
class HelloController < ApplicationController
end
```

And as you've also already seen, you create actions simply by adding methods to the controller:

```
class HelloController < ApplicationController
    def there
    end
end
```

The controller calls the various actions, and when the actions have done their thing, the controller passes the results of the application on to a view.

The View

A view is responsible for displaying the results of an action. There can be many different views in a web application, because web applications can display many different pages in the user's browser. The view you've seen so far has been static, but of course you usually want to display data in the view. That means that an action will typically pass data on to the view, and you're going to see how to do that in this chapter. Rails view templates often allow you to insert data into them before they are sent back to the user's browser by the controller.

In other words, you use views to interact with the user. When you want to read data from the user, you send a view with various HTML controls such as text fields, list boxes, text areas, and so on. When the user clicks the Submit button, that data is passed to your application's controller, which hands it off to an action, which in turn passes it to a view that is sent back to the browser.

Views are supported with the ActionView modules in Rails. In fact, in Rails, controllers and views are so tightly integrated that together, the ActionView and ActionController modules are referred to as the ActionPack. So the controller routes requests to actions, and the actions send views to the user's browser.

The Model

You also need the *model* in a Rails application. The model handles the data processing that takes place in a web application. Actions can interact with the model to handle the data churning that needs to be done.

For example, the model is where you can place the business rules of an application that figure the tax and/or shipping on an order from the user. Or you can check a database to see whether an item is in stock. Or you can look up a user's information from another database. That is, the model is the number cruncher in the application. It has no clue about its environment, and

knows nothing about being on a web server as part of an online application. You just hand it data and tell it what to do. It does the data-handling work, and returns the result. Typically, actions pass data into the model and then retrieve the results.

Rails is written especially to handle databases, and the model is where that support is. You can base models on the Rails ActiveRecord module—note, not ActionRecord, but ActiveRecord. That's the overview of the model-view-controller architecture that Rails uses. Because Rails applications are broken up into these components, it's important to know what they do. You've already seen the controller and view at work in your first Rails web application, and you're going to see how to work with models soon.

So far, the view has been pretty static, just displaying a welcome message. It's time to add some more functionality there.

5.5.6.Connecting to Databases

Web applications often store data online on a server. And most often, they use databases to store that information. Ruby on Rails is especially built to handle databases online easily.

Tutorial on Databases : The following table 1 shows students grades.

Table 1: Students

Name	Grade	ID
Tom	A	1
Carol	B	2
Frank	B	3
Anne	A	4
Sam	A	5
Nancy	B	6
Pat	C	7

This is actually a paper version of what is called a *table* in a database. Data of Name, Grade, and ID in the students table are called columns. ach student gets a *record* in the table. A record is a row in the table, and it contains all of the columns' information for a particular student: name, grade, and ID. That is you create a table in a database simply by putting together the columns and rows of that table. (The intersection of a column and a row is called a *field*; a row in this example has three fields: Name, Grade, and ID.).

Databases can contain many tables, and in a *relational* database, you can relate those tables together. For example, you may also want to keep track of how much money each student owes you in addition to the students table information. This new table might be called fees, and it might look like

ID	Owes
1	20,005.00
2	19,005.00
3	23,005.00
4	21,005.00
5	22,005.00
6	20,005.00
7	21,005.00

The fees table keeps track of the amount each student owes by ID. If you wanted to know how much the student named Tom owed, but didn't have his ID handy, you could look up Tom in the students table, find his ID, and use it in the fees table. In other words, the records in the students and fees tables are tied together by the ID field, as shown in the figure.

Name	Grade	ID		ID	Owes
Tom	A	1	⟷	1	20,005.00
Carol	B	2	⟷	2	19,005.00
Frank	B	3	⟷	3	23,005.00
Anne	A	4	⟷	4	21,005.00
Sam	A	5	⟷	5	22,005.00
Nancy	B	6	⟷	6	20,005.00
Pat	C	7	⟷	7	21,005.00

You can store these two tables, students and fees, in a single database, and relate them, record by record, using the ID field. A field that you use to connect records in a table to records in another table is called a *primary key*. The ID field in the students table is that table's primary key. The primary key is the main data item you use to index records in a table. You're going to need an ID column in tables you use with Rails applications—actually, Rails needs id, Each table should have a column named id.

To construct an online store, you need two controllers—the management controller used to update the online database with what's in stock and set prices, and the customer controller used to let people buy from the store. Begin by creating the store application itself. This application is placed in the msc directory:

```
C:\rubydev\msc>rails store
```

OK, that gives you the application framework.

Creating the Database

After installing MySQL, to create the database for the store application to use, start the MySQL monitor on the command line like this:

```
C:\rubydev\msc>mysql -u root -p
```

This command gives root as the username and tells the MySQL monitor to ask for a password. Enter the password you set during MySQL installation at the prompt:

```
C:\rubydev\ch06>mysql -u root -p
Enter password: *********

Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 5.0.19-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql>
```

Your goal is to create a database to hold the items for sale in your online store. In fact, establish three databases—one for development, one for testing, and one for production. Name them store_development, store_test, and store_production, and create them using the create database command in the MySQL monitor:

```
mysql> create database store_development;
    Query OK, 1 row affected (0.06 sec)
mysql> create database store_test;
    Query OK, 1 row affected (0.01 sec)
mysql> create database store_production;
    Query OK, 1 row affected (0.00 sec)
```

To make MySQL work with that database, enter the use command and the name of the database to use, store_development:

```
mysql> use store_development
    Database changed
mysql>
```

Now create a table in the store_development database. This table will hold the items for sale in your online store, so a good name for this table is simply items. You can create a new table named items in the store_development database with the create table command:

```
mysql> create table items (
-> id int not null auto_increment,
-> name varchar(80) not null,
-> description text not null,
-> price decimal(8, 2) not null,
-> primary key(id)
-> );
```

MySQL responds with:
Query OK, 0 rows affected (0.06 sec)

exit MySQL:

```
mysql> exit
Bye
```

Configuring Database Access

In the rubydev\msc\store\config directory, you'll find a file named database.yml. It lets you connect your application to your database.

```
# MySQL (default setup). Versions 4.1 and 5.0 are recommended.
#
# Install the MySQL driver:
# gem install mysql
# On MacOS X:
```

```
# gem install mysql -- --include=/usr/local/lib
# On Windows:
# There is no gem for Windows. Install mysql.so from RubyForApache.
# http://rubyforge.org/projects/rubyforapache
#
# And be sure to use new-style password hashing:
# http://dev.mysql.com/doc/refman/5.0/en/old-client.html
development:
adapter: mysql
database: store_development
username: root
password:
host: localhost
# Warning: The database defined as 'test' will be erased and
# re-generated from your development database when you run 'rake'.
# Do not set this db to the same as development or production.
test:
adapter: mysql
database: store_test
username: root
password:
host: localhost
production:
adapter: mysql
database: store_production
username: root
password:
host: localhost
```

Creating the Controller and Model

Actually connecting the database to your code could be tough, but Rails has a utility named scaffold that makes the process easy. The purpose of scaffold is much as it sounds: to create a scaffold for your application—that is, a framework that you can fill in, or not, as you decide. You can use the scaffold utility to build a model and controller for data-aware applications. Here's how you'd do so:

1. Change directories to the rubydev\ch06\store directory:

```
C:\>cd \rubydev\msc\store>ruby
C:\rubydev\msc\store>
```

2. Create a model named Item and a controller named Manage this way:

```
C:\rubydev\msc\store>ruby script/generate scaffold Item Manage
```

3. Rails creates the controller and model:

```
C:\rubydev\msc\store>ruby script/generate scaffold Item Manage
exists app/controllers/
exists app/helpers/
create app/views/manage
exists test/functional/
```

dependency model
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/item.rb
create test/unit/item_test.rb
create test/fixtures/items.yml
create app/views/manage/_form.rhtml
create app/views/manage/list.rhtml
create app/views/manage/show.rhtml
create app/views/manage/new.rhtml
create app/views/manage/edit.rhtml
create app/controllers/manage_controller.rb
create test/functional/manage_controller_test.rb
create app/helpers/manage_helper.rb
create app/views/layouts/manage.rhtml
create public/stylesheets/scaffold.css

Rails' scaffold utility builds the application's model and controller. The model is named item.rb, and it contains a new class named Item, inherited from ActiveRecord::Base:

```
class Item < ActiveRecord::Base
End
```

ActiveRecord is the module that contains the Rails support for working with databases, and the Base class is the primary class on which you base database-aware models.

Naming the Model

The name of the model, Item, was specially chosen. As you may recall, the database created for the store application was named store_development, and the table contained inside that database was named items. The two names—Item and items—are tied together.

Naming the Controller

You can choose the name of the controller with more freedom. In the example application, the controller is to be used when managing the online store—adding new items for sale, listing the prices, and so on, so the controller is simply named Manage. That creates a controller class named ManageController, inherited from ApplicationController, just as the controllers you saw earlier. However, scaffold stocks this controller with code, as you see here in manager_controller.rb:

```
class ManageController < ApplicationController
def index
list
render :action => 'list'
end
# GETs should be safe (see http://www.w3.org/2001/tag/doc/whenToUseGet.html)
verify :method => :post, :only => [ :destroy, :create, :update ],
:redirect_to => { :action => :list }
def list
```

```
@item_pages, @items = paginate :items, :per_page => 10
end
def show
@item = Item.find(params[:id])
end
def new
@item = Item.new
end
def create
@item = Item.new(params[:item])
if @item.save
flash[:notice] = 'Item was successfully created.'
redirect_to :action => 'list'
else
render :action => 'new'
end
end
def edit
@item = Item.find(params[:id])
end
def update
@item = Item.find(params[:id])
if @item.update_attributes(params[:item])
flash[:notice] = 'Item was successfully updated.'
redirect_to :action => 'show', :id => @item
else
render :action => 'edit'
end
end
def destroy
Item.find(params[:id]).destroy
redirect_to :action => 'list'
end
end
```

A number of actions have already been built into the Manage controller. How about taking a look at them in action?

Running the store Application

Change directories to the rubydev\msc\store directory and run WEBrick to start the store application like this:

```
C:\rubydev\ch06\store>ruby script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
```

[2006-05-30 11:23:26] INFO WEBrick 1.3.1
[2006-05-30 11:23:26] INFO ruby 1.8.2 (2004-12-25) [i386-mswin32]
[2006-05-30 11:23:26] INFO WEBrick::HTTPServer#start: pid=1252 port=3000

Now navigate to http://localhost:3000/manage, as shown in the following Figure. You see the titles of the fields in the items table—Name, Description, and Price. In other words, the model was successful in connecting to the items database table. (The id field isn't shown because it's an auto-increment field.)

By entering simply the name of the controller, without an action, in the URL (http://localhost: 3000/manage), you called the default index action, which looks like this in the manage_controller.rb file:

```
class ManageController < ApplicationController
    def index
        list
        render :action => 'list'
    End
```

This method calls the list action, which loads the current items in the items table into item_pages and @items:

```
        def list
            @item_pages, @items = paginate :items, :per_page => 10
        End
```

The index action then calls render :action => 'list' to render the view associated with the list action, which is rubydev\msc\store\app\views\manage\list.rhtml. The list view lists the current records in the items table. However, there are no records to display yet. Change that by adding some.

Adding a Record to the store Application

The New Item link links to http://localhost:3000/manage/new, bringing up to create new records by entering data into various HTML controls—a text field for the name of the new item you're creating in the store's database, a text area for its description, and so on.

The New Item link accesses the new action to display the page, which creates a new record using the model class, Item:

```
def new
    @item = Item.new
end
```

that's the method that lets you display a view in the browser. When you render a partial view, you render it in-place—that is, inside the current view. So render :partial => 'form' really means "display the form view at this location in the current view."

When you fill out the controls in this view with the data for the new record and click the Create button, you navigate to the create action, which looks like this in the manage controller:

```
def create
    @item = Item.new(params[:item])
    if @item.save
        flash[:notice] = 'Item was successfully created.'
        redirect_to :action => 'list'
    else
        render :action => 'new'
    end
end
```

This is where the new record is created. In the create action, Rails uses the information entered into the _form view to create a new model object like this:

```
def create
    @item = Item.new(params[:item])
    .
    .
    .
end
```

Then the create action attempts to save the new record in the database, using the model's save method (built into ActiveRecord::Base, the class from which the Item class inherits) this way:

```
def create
    @item = Item.new(params[:item])
    if @item.save
        .
        .
        .
    end
end
```

If the save operation is successful, the save method returns a value of true, and the create action executes this code:

```
def create
    @item = Item.new(params[:item])
    if @item.save
        flash[:notice] = 'Item was successfully created.'
        redirect_to :action => 'list'
        .
        .
```

```
            .
        end
    end
```

In this case, the new item has been successfully created, so the message passed on to the list action is 'Item was successfully created.'. If the new item had not been successfully created, the code renders the new action so you can try to create the item again:

```
    def create
        @item = Item.new(params[:item])
        if @item.save
            flash[:notice] = 'Item was successfully created.'
            redirect_to :action => 'list'
        else
            render :action => 'new'
        end
    end
```

Displaying a New Record

It's possible because this application uses a Rails *layout*. Layouts are templates that views are inserted into automatically. Using a layout gives your application a consistent feel across many different views, because you can create standard headers and footers for each web page, as well as use a standard stylesheet, and more.

Using a Layout

The layout for all the actions in the manage controller in the store application is in the file manage.rhtml, stored as rubydev\store\msc\app\views\layouts\manage.rhml. In manage.rhtml, the contents of the @content_for_layout variable are simply popped into the web page like this:

```
<%= @content_for_layout %>
```

Using a Stylesheet

Note the stylesheet_link_tag method in manage.rhtml, which links to the stylesheet scaffold.css:

```
<%= stylesheet_link_tag 'scaffold' %>
```

This stylesheet, rubydev\store\ch06\public\stylesheets\scaffold.css, specifies the cascading style sheet (CSS) styles used in the layout, and, therefore, in all the views displayed using the layout.

Displaying Records

The list view displays all the records in the items table in an HTML table. That HTML table is created using embedded Ruby to call the Item model's content_columns class method to get the names of the columns in the items table, and to display them by calling each column object's human_name method (which, in this case, just returns the column's name in the items table):

To see the contents of the items table, enter the SQL command select * from items;, which selects and displays all records in the items table.

Adding Another Record

To create another new record, just navigate to http://localhost:3000/manage. The manage controller's default action is the index action, which renders the list action like this in the manage controller:

```
def index
    list
    render :action => 'list'
end
```

To create a new record, follow these steps:

1. Start the WEBrick server:

```
C:\rubydev\msc\store>ruby script/server
```

2. Navigate to http://localhost:3000/manage.
3. In the list view, click the New Item link to get to the http://localhost:3000/manage/newpage.
4. Create a new item for sale, giving it the name clock, the description "A nice digital clock. Contains all you would expect—and the alarm is not too loud!", and the Click the Create button on the page (http://localhost:3000/manage/new). The new item is created and your browser is redirected to the http://localhost:3000/manage/list page.

Editing Records

Making changes to records is no problem with the manage controller—just click the Edit link. In the following exercise you update the price of alarm clocks in your database.

To edit a record, follow these steps:

1. Start the WEBrick server:

```
C:\rubydev\msc\store>ruby script/server
```

2. Navigate to http://localhost:3000/manage/.
3. Click the Edit link for the alarm clock.
4. Change the price of the alarm clock to $15.99.
5. Click the Edit button.

   The alarm clock's record is updated, and the new data is displayed.

6. Click the Back link at the bottom of the clock record to return to list view. As you can see in that view, the price of the alarm clock has indeed been updated.

Beautifying the Display

Bear in mind that you have control over what all these views look like—the scaffold utility generates template files, and it's up to you to customize them as you want. For example, you might want to beautify the display of the list view. To do so, you could follow these steps:

1. Add background0 and background1 style classes to rubydev\msc\store\public\stylesheets\scaffold.css:

```
body { background-color: #fff; color: #333; }
.
.
.
.fieldWithErrors {
    padding: 2px;
```

```
        background-color: red;
        display: table;
}
.background0 {
        background-color: coral;
}
.background1 {
        background-color: white;
}
.
.
.
```

2. Modify rubydev\msc\store\app\views\manage\list.rhtml, adding this code:

```
<h1>Listing items</h1>
<table>
<tr>
<% for column in Item.content_columns %>
<th><%= column.human_name %></th>
<% end %>
```

Now start the WEBrick server and navigate to http://localhost:3000/manage/list. The page shows the new list view with a number of additional records.

5.6.Working with Databases

Let us consider the store application example from rubydev\msc\store to display your items to users, and to get their purchases in a cart.

5.6.1.Displaying Items to the Customer

The manage controller is used to control the store application's connection to the database. You need a second controller if you want to let users peruse the store—obviously, they shouldn't have access to the administrative part of the store. The following exercise leads you through the creation of another controller.

To create a second controller for the store application, follow these steps:

1. Change directories to the rubydev\ch07\store directory:

```
C:\>cd \rubydev\ch07\store>ruby
C:\rubydev\ch07\store>
```

2. Create the second controller, named buy, like this:

```
C:\>cd \rubydev\ch07\store>ruby script/generate controller Buy index
```

3. Start the WEBrick server:

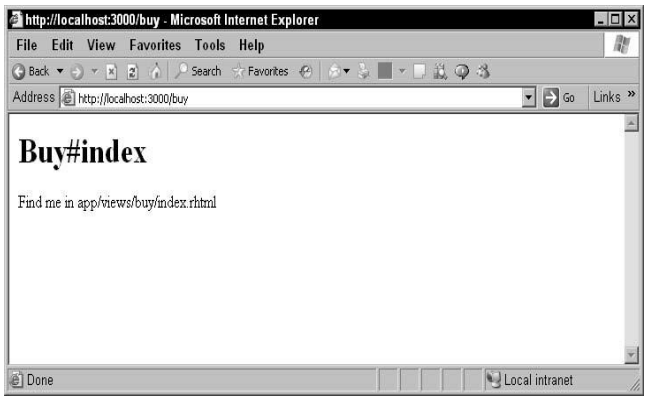
```
C:\rubydev\ch07\store>ruby script/server
```

4. Navigate to http://localhost:3000/buy.

The command

C:\>cd \rubydev\ch07\store>ruby script/generate controller Buy index

creates the buy controller and the action named index. Because you've named the action index, it'll be the default action, executed if the user navigates to http://localhost:3000/buy.



5.6.2.Getting the Items for Sale

How do you get the records from the items table?

You can create a class method in the model, rubydev\ch07\store\app\models\item.rb, named, say, return_items, like this:

```
class Item < ActiveRecord::Base
    def self.return_items
    .
    .
    .
    end
end
```

How does that work?

You call find(:all) in return_items in the model class, rubydev\msc\store\app\models\item.rb:

```
class Item < ActiveRecord::Base
    def self.return_items
        find(:all)
    end
end
```

There are actually three ways to call find:

- ✓ find(:id)—Finds a record by ID
- ✓ find(:first)—Finds the first record
- ✓ find(:all)—Returns all the records in the table

In fact, you can specify other requirements in hash form following :id, :first, or :all. Here's an example that finds items less than or equal to $20.00, and orders the found records by name, and if the name is the same, by description:

```
class Item < ActiveRecord::Base
    def self.return_items
        find(:all,
        :order => "name description",
        :conditions => "price <= 20.00"
        )
    end
end
```

Ruby	SQL Information
:conditions	SQL code indicating a condition or conditions to match.
:group	Specifies an attribute indicating how the result should be grouped, making use of the SQL GROUP BY clause.
:include	Specifies associations to be included using SQL LEFT OUTER JOINs.
:joins	Specifies additional SQL joins.
:limit	Specifies an integer setting the upper limit of the number of rows to be returned.
:offset	Specifies an integer indicating the offset from where the rows should be returned.
:order	Lets you specify the fields to set the order of returned records.
:readonly	Marks the returned records read-only.
:select	A SQL SELECT statement, as in SELECT * FROM items.

OK, a new class method in the Item model, return_items, returns the full set of records in the items table. To start the process of displaying the items for sale when the buy controller's index action is called, you call return_items in the index action:

```
class BuyController < ApplicationController
        def index
                @items = Item.return_items
        end
end
```

Making progress load all the records from the items table into the @items array. Now display them.

5.6.3.Showing the Items for Sale

Displaying the records is the job of the view connected to the index action, rubydev\ch07\store\app\views\buy\index.rhtml, which currently looks like this:

```
<h1>Buy#index</h1>
<p>Find me in app/views/buy/index.rhtml</p>
```

Show Database Items in a Web Page

To show the database items stored in the @items array in a web page, follow these steps:

1. Create a new web page:

```
<html>
<head>
<title>The Store</title>
</head>
<body>
<h1>Buy From Our Store!</h1>
<b>Welcome to the store.</b>
<br>
<b><i>Please buy a lot of items, thank you.</i></b>
```


.
.
.
</body>
</html>
2. You can display the items for sale in an HTML table, so add that as well:
<html>
<head>

<title>The Store</title>
</head>
<body>
<h1>Buy From Our Store!</h1>
Welcome to the store.

Please buy a lot of items, thank you.

<table cellpadding=”6”>
.
.
.
</table>
</body>
</html>
3. Add a loop over the items in the @items array:
<html>
<head>
<title>The Store</title>
</head>
<body>
<h1>Buy From Our Store!</h1>
Welcome to the store.

Please buy a lot of items, thank you.

<table cellpadding=”6”>
<% for item in @items %>
.
.
.
<% end %>

```
</table>
</body>
</html>
```

4. Display the name and description of each item, row by row in the table:

```
<table cellpadding=”6”>
<% for item in @items %>
<tr>
<td><b><%=h item.name %></b></td>
<td><%=h item.description %></td>
.
.
.
</tr>
<% end %>
</table>
```

Finally, add a link that lets the user add each item to the shopping cart:

```
<table cellpadding=”6”>
<% for item in @items %>
<tr>
<td><b><%=h item.name %></b></td>
<td><%=h item.description %></td>
<td><%= link_to ‘Add to cart’, :action => ‘add’, :id => item %></td>
</tr>
<% end %>
</table>
```

6. Start the WEBrick server:

```
C:\rubydev\ch07\store>ruby script/server
```

7.Navigate to http://localhost:3000/buy to see the result, as shown in Figure.

That finishes the display of the items for sale. You can beautify the display by using CSS formatting, and work on polishing the HTML for the item display. How to grab data from a database table and display it.

Of course, the purpose of displaying the items is so users will buy them. You need to start building a shopping cart so that can happen.

5.7. Creating a Shopping Cart

Users are presented with all items in the store in the index.rhtml view. If they want to buy an item, they click the Add to Cart link that you created in step 5 of the preceding

```
<tr>
<td><b><%=h item.name %></b></td>
<td><%=h item.description %></td>
<td><%= link_to 'Add to cart', :action => 'add', :id => item %></td>
</tr>
```

That link connects to the add action in the buy controller

5.7.1.Designing the Shopping Cart

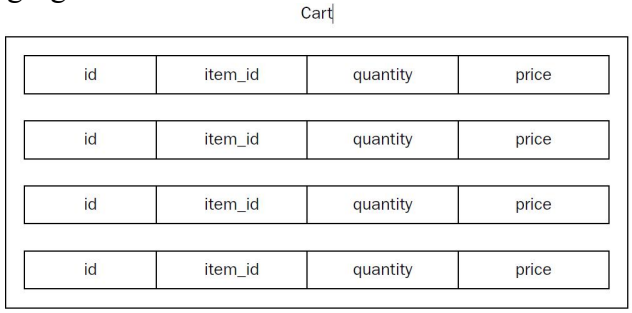
The session hash is going to be involved with the shopping cart because it's a good place to store the customer's purchases. Theoretically, you could just set up some arrays to store the names of the items the customer has purchased, their descriptions, and their prices, and store that in the session hash.

Then when you need that data, you simply fetch it from the session. That's certainly one way to handle the shopping cart data. The data handled by a model in Rails applications. Rails data-handling is model-centric, and your code is made easier if you use models (for instance, you can access all your data as @model_object.name, @model_object.id, @model_price, and so on). Storing data with a model in this example will also point out some new aspects of database-handling in Rails.

Here's the plan: create a new model named Purchase to hold a single item that the customer has purchased.

The model will be tied to a table named purchases in the store_development, with the following fields: id, item_id (this is the item's id in the items table), quantity, and price.

Store a single purchase as a record in the purchases table, the customer will buy many items, store them in an array in an object of a new class, the Cart class, as shown in the following figure.



So you need two new classes—the Purchase class to keep track of individual purchases, and the Cart class to keep track of the many Purchase objects a customer creates as he shops.

Creating the purchases Table

The Purchase class is a model tied to a database table, the purchases table, so first create that table. Just follow these steps:

1. Start the MySQL monitor:

```
C:\rubydev\ch07\store>mysql -u root -p
.
.
mysql>
```

2. Switch to the store_development database:

```
mysql> use store_development
Database changed
```

3. Create the purchases table:

```
mysql> create table purchases (
-> id int not null auto_increment,
-> item_id int not null,
-> quantity int not null default 0,
-> price decimal(8, 2) not null,
-> constraint purchases_items foreign key (item_id) references items(id),
-> primary key (id)
-> );
Query OK, 0 rows affected (0.05 sec)
mysql>
```

4. Exit the MySQL monitor:

```
mysql> exit
Bye
```

The items table stores the items for sale. Like the purchases table, the items table has an id value for each record. The preceding line of SQL says that the item_id field in the purchases table holds the id value of the item in the items table. That is, item_id is a foreign (not primary) key. Foreign keys can be used to relate tables—if you want to look up a purchase in the items table, for instance, you can do that using the item_id value in the purchases table.

Creating the Purchase Model

The purchases table, each record of which will hold a purchase by the customer, now needs to be tied to a model in the store application. To do that, change directories to rubydev\msc\store, and then create the new model like this:

```
C:\rubydev\ch07\store>ruby script/generate model Purchase
```

Now when the user makes a purchase, you'll catch that item, create a Purchase object from it, and then store that object in the cart. Here's the new model, purchase.rb:

```
class Purchase < ActiveRecord::Base
End
```

Still have to tell Rails about the foreign key connection to the items table:

```
mysql> create table purchases (
-> id int not null auto_increment,
```

```
-> item_id int not null,
-> quantity int not null default 0,
-> price decimal(8, 2) not null,
-> constraint purchases_items foreign key (item_id) references items(id),
-> primary key (id)
-> );
```

The Rails core development team set things up so that you have to tell Rails about foreign keys yourself.cThat's because not all database servers let you work with foreign keys. You tell Rails about yourcitem_id foreign key with the belongs_to method in the purchase.rb. model. The syntax looks like this:

```
class Purchase < ActiveRecord::Base
belongs_to :item
.
.
.
End
```

That connects the purchases and items tables as far as Rails is concerned. You're going to need the belongs_to method each time you want to connect database tables using foreign keys. You also need some way of producing new Purchase objects when the user buys something.

- ✓ Create a new class method named buy_one that you can pass an item (from the items table) to when the user purchases that item:
- ✓ So when the user buys a displayed item, you can pass that item to the Purchase class's buy_one method to create a Purchase object from that item. That means that when you call the buy_one method, you start by creating a new Purchase object:
- ✓ To make life easier for yourself, you can store the purchased item inside the new Purchase object.
- ✓ Now when you create a new Purchase object, you'll have access to the actual Item object that was purchased, which means you can access its name, description, and so on.
- ✓ Next, you can store the quantity of the item purchased in the Purchase object, enabling the user to purchase multiple items.
- ✓ You also create a new field for the Purchase object named price, holding the price of the item.
- ✓ Finally, you just return the new Purchase object from the buy_one method:

```
class Purchase < ActiveRecord::Base
        belongs_to :item
        def self.buy_one(item)
                purchase = self.new
                purchase.item = item
                purchase.quantity = 1
                purchase.price = item.price
                return purchase
        end
end
```

OK, the Purchase class is ready to go—when the user buys one of the displayed items, all you have to do is to fetch the item from the items table and pass it to the Purchase class's

buy_one method to create a new Purchase object. That new Purchase object should go into the shopping cart—which means you've got to create a Cart class next.

Creating the Cart

The Cart class exists to keep track of the purchases the user makes—which means keeping track of Purchase objects as they're created.

- ✓ Start writing the Cart class, by creating an empty Purchases array named @purchases in the initialize constructor:
- ✓ The cart should also keep track of the total price of all the purchased items, so you might add an attribute named price to the Cart class, setting it to 0.0 when the cart is first created.
- ✓ You can make the purchases and price attributes available publicly by using attr_reader.
- ✓ Next, you're going to need a way to add new purchases to the cart. A convenient means to do that is to add a new method, add_purchase, to the Cart class. It's easiest to set this method up to accept items as the user purchases.
- ✓ When you pass a new item to the add_purchase method, you can use that item to create a new Purchase object, and add that new object to the purchases array using the array append operator, <<.
- ✓ And when you add a new purchase to the cart, update the total attribute of the cart by adding the price of the new item to the total.

```
class Cart
        attr_reader :purchases
        attr_reader :total
        def initialize
                @purchases = []
                @total = 0.0
        end
        def add_purchase(item)
                @purchases << Purchase.buy_one(item)
                @total += item.price
        end
end
```

That's the Cart class, which is stored in cart.rb. You can place it with the other models in rubydev\msc\store\app\models\cart.rb.The Cart class is not derived from the ActiveRecord::Base class, but it's as much a model as the other two models in this example (item.rb and purchase.rb).

Storing the Cart in a Session

As the customer navigates from page to page, the server is going to lose control of the application, which means all your data will be reset to its initialization values. To store the purchases in the cart, you've got to store the whole cart in the session. To make accessing the cart easy, create a helper method in the controller, get_cart:

- ✓ This helper method is private, which means Rails won't make it into an action. Private methods in the controller stay private to the controller and are not accessible as public actions.
- ✓ When get_cart is called, it should first check if the cart was already stored in the session and if so, it should return the cart:

- ✓ If the cart doesn't exist in the session, the get_cart method should create a new Cart object and return that:

```
class BuyController < ApplicationController
    def index
        @items = Item.return_items
    end
private
    def get_cart
        if session[:shopping_cart]
            return session[:shopping_cart]
        else
            return Cart.new
        end
    end
end
```

The application.rb file is run first so you can initialize your application. To make the store application pre-load the Cart and Purchase classes, add the following code to rubydev\ch07\store\app\controllers\application.rb:

```
class ApplicationController < ActionController::Base
    model :cart
    model :purchase
    end
```

The next step is to put the cart to use and handle a purchase.

Handling a Purchase

When the user navigates to the store, http://localhost:3000/buy, he sees the buy page that displays the items available. When the user clicks an item's Add to Cart link, that item's ID is sent to the controller. Now you define a new action named add to tell the controller it's time to add that item to the cart.

To add a purchase to the cart, follow these steps:

1. Edit the rubydev\ch07\store\app\controllers\buy_controller.rb, adding the add action:

```
class BuyController < ApplicationController
    def index
        @items = Item.return_items
    end
    def add
    .
    .
    end
Private
    def get_cart
        if session[:shopping_cart]
            return session[:shopping_cart]
        else
            return Cart.new
        end
```

```
        end
    end
```

2. Add this code to the add action:

```
    def add
        item = Item.find(params[:id])
        @cart = get_cart
        @cart.add_purchase(item)
        session[:shopping_cart] = @cart
        redirect_to(:action => 'display_cart')
    end
```

Displaying the Cart

The display_cart action displays the cart, so add that action to rubydev\msc\store\app\controllers\buy_controller.rb now:

```
class BuyController < ApplicationController
    def index
        @items = Item.return_items
    end
    def display_cart
        .
        .
    end
private
    def get_cart
        if session[:shopping_cart]
            return session[:shopping_cart]
        else
            return Cart.new
        end
    end
end
```

display_cart simply needs to set up the data to be displayed in the accompanying view, so you start by getting the cart from the session, using the get_cart method, and load two instance variables: @purchases with the array of purchases and @total with the total of the purchases, like this:

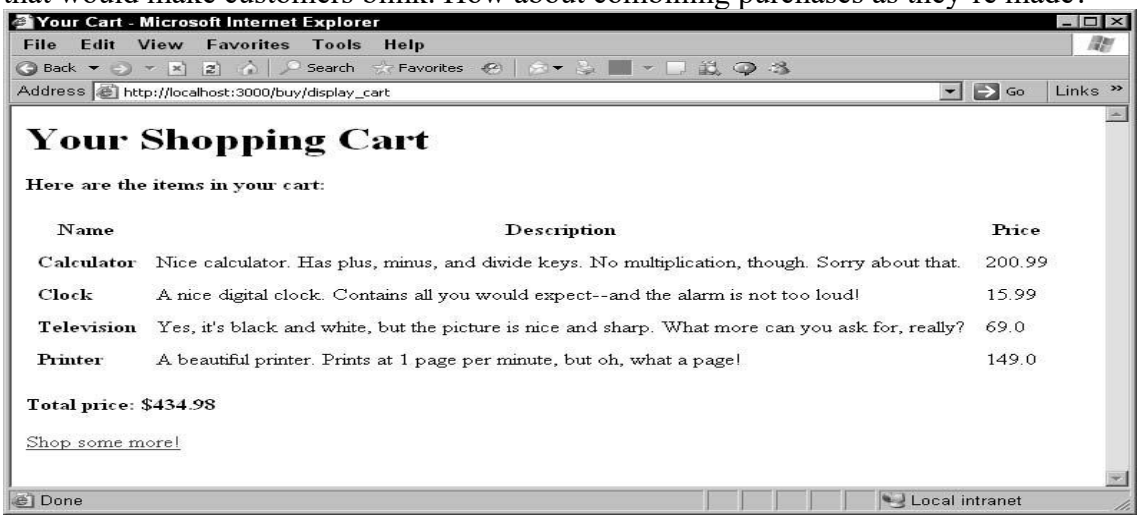
```
def display_cart
    @cart = get_cart
    @purchases = @cart.purchases
    @total = @cart.total
end
```

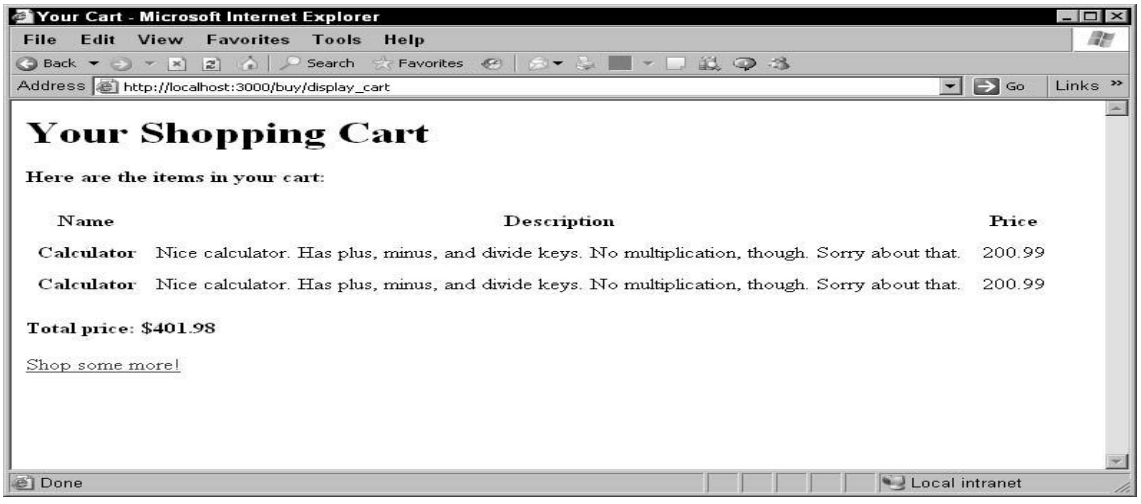
That makes the data in the cart accessible to the display_cart view. In that view, rubydev\msc\app\views\buy\display_cart.rhtml, you need to display both the purchases the user made and the total.

The first figure shows the result after the user makes some purchases by clicking the Add to Cart link. As you see, the purchased items appear in the cart, and the total cost of all the items appears at the bottom of the page.

Not bad—you've now been able to let the customer select items to buy, and have displayed his shopping cart as he adds items to it.

But what if a customer wants to purchase *two* calculators? You'd end up with the situation shown in the next Figure, where you have two single entries for calculators—clearly, that would make customers blink. How about combining purchases as they're made?





Combining Purchases in the Cart

To combine purchases in the cart, you have to modify cart.rb—specifically, the add_purchase method, which simply (and naively) adds new items to the @purchases array:

```
def add_purchase(item)
        @purchases << Purchase.buy_one(item)
        @total += item.price
end
```

- ✓ You need to check whether the item being added to the @purchases array is already in that array. To do so, loop over the array.
- ✓ It's easy enough to compare the ID of the new item to the items already in the @purchases array, this way.
- ✓ If the item already exists in the @purchases array, you can set a true/false flag, appendFlag, to false, indicating that the item should not be appended to the array.
- ✓ If the item already exists in the @purchases array, all you need to do is to increase the quantity of the item by one:
- ✓ If the item doesn't already exist in the array, you create a new element in @purchases. In either case, you update the total with the new item's price, as before:

```
def add_purchase(item)
        appendFlag = true
        for purchase in @purchases
                if (item.id == purchase.item.id)
                        appendFlag = false
                        purchase.quantity += 1
                end
        end
        if(appendFlag)
                @purchases << Purchase.buy_one(item)
        end
        @total += item.price
end
```

Now the code does the right thing with the @purchases array when the customer purchases multiple items. To display that array correctly, you need to modify display_cart.rhtml. Start by adding a new column to the display of purchased items—the quantity of each item purchased. And there you have it—now when the customer buys several of the same items, the store application handles the situation correctly by displaying the quantity of each item in the display_cart.rhtml view will shown in the following Figure.

Clearing the Cart

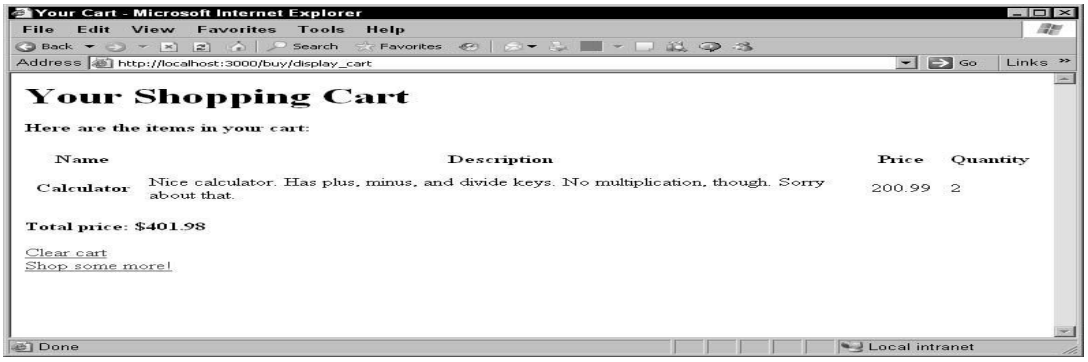
Let the customer clear the cart when he wants to. To do that, add a link in the display_cart.rhtml file with the text Clear cart, connected to the action clear_cart: You can see this new link in the display cart page in following figure. When the user clicks that link, control is transferred to the clear_cart action, which calls the initialize method of the Cart object to clear the cart:

```
class BuyController < ApplicationController
        def index
                @items = Item.return_items
        end
        .
        .
        def clear_cart
                @cart = get_cart
                @cart.initializeinitialize
        end
```

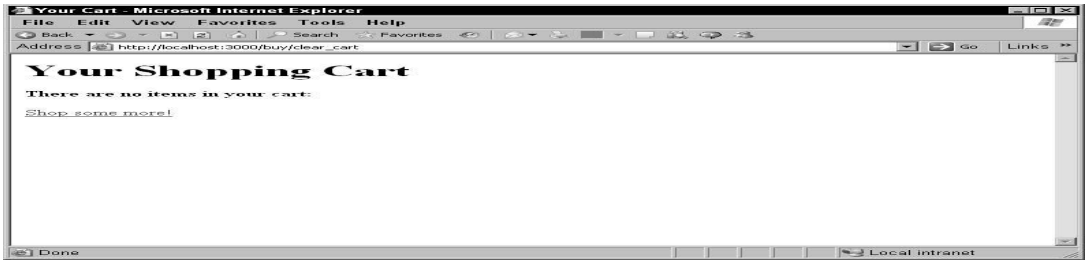
```
private
    def get_cart
        if session[:shopping_cart]
            return session[:shopping_cart]
        else
            return Cart.new
        end
    end
end
```

In cart.rb, add the clear method to reset the @purchases array and set the current total price to 0.0:

```
    class Cart
        attr_reader :purchases
        attr_reader :price
        def initialize
            @purchases = []
            @price = 0.0
        end
        def clear
            @purchases = []
            @price = 0.0
        end
        .
        .
        end
```



All that's left is to tell the customer that the cart has indeed been cleared. You can do so by displaying a new page, clear_cart.rhtml, connected to the clear_cart action, like this: The following figure shows the page that comes up after the user clears the cart.



Letting the User View the Cart Anytime

As a final refinement, let the user view his cart at anytime. To do that, just add a link, See your cart, to the index.rhtml page that displays the items for sale:

```
<html>
	<head>
		<title>The Store</title>
	</head>
	<body>
		<h1>Buy From Our Store!</h1>
		<b>Welcome to the store.</b>
		<br>
		<b><i>Please buy a lot of items, thank you.</i></b>
		<br>
		<br>
		<table cellpadding=”6”>
			<% for item in @items %>
			<tr>
				<td><b><%=h item.name %></b></td>
				<td><%=h item.description %></td>
				<td><%= link_to ‘Add to cart’, :action => ‘add’, :id => item
				%></td>
			</tr>
			<% end %>
		</table>
		<br>
		<%= link_to ‘See your cart’, :action => ‘display_cart’ %>
	</body>
</html>
```

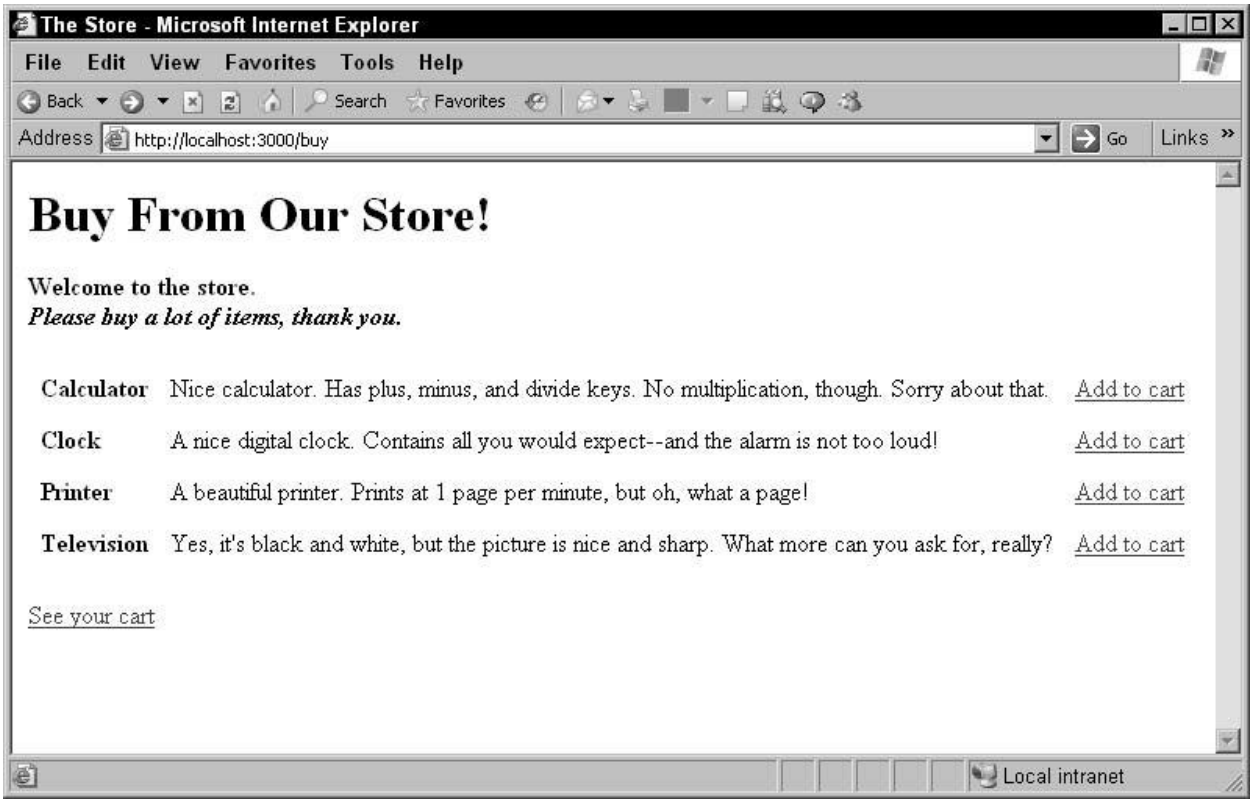


Figure shows the new link on the page.

Letting the customer view the cart at any time means that the cart could be empty, so it'd be a problem trying to loop over the @purchases array in the display_cart.rhtml view. To head that off, check to see if @purchases is empty in display_cart.rhtml, and if so, display a message that the cart is empty, like this:

```
<html>
    <head>
        <title>Your Cart</title>
    </head>
    <body>
        <h1>Your Shopping Cart</h1>
        <% if (@purchases == []) %>
        <b>There are no items in your cart:</b>
        <br>
        <br>
        <%= link_to 'Shop some more!', :action => 'index' %>
        <% else %>
        <b>Here are the items in your cart:</b>
        <br>
        <br>
        <table cellpadding="6">
        <tr>
        <% for column in Item.content_columns %>
```

```
                <th><%= column.human_name %></th>
                <% end %>
                <th>Quantity</th>
            </tr>
            <% for purchase in @purchases
            item = purchase.item
            %>
            <tr>
                <td><b><%=h item.name %></b></td>
                <td><%=h item.description %></td>
                <td><%=h item.price %></td>
                <td><%=h purchase.quantity %></td>
            </tr>
            <% end %>
            </table>
            <br>
            <b>Total: $<%=h @total %></b>
            <br>
            <br>
            <%= link_to 'Clear cart', :action => 'clear_cart' %>
            <br>
            <%= link_to 'Shop some more!', :action => 'index' %>
            <% end %>
        </body>
</html>
```

And that's it—you've created a full, multi-page shopping cart demonstration application.

EXERCISES

1. Use a negative index to access the third element in this array: array = [1, 2, 3, 4, 5, 6, 7, 8].
2. Construct a hash that will act the same as the array introduced in the previous exercise, as far as the [] operator is concerned.
3. Use a range to create the array introduced in exercise.
4. Create a method that calls itself—a technique called *recursion*—to calculate *factorials*. A factorial is the product of the number times all the other whole numbers down to one—for example, the factorial of 6, written as 6!, is 6 × 5 × 4 × 3 × 2 × 1 = 720.
5. Construct a method named printer that you can call with some text, which then calls a code block to print out that text.
6. Create a method named array_converter that takes four arguments and returns them in an array. Create a class named Vehicle, and pass the color of the vehicle to the constructor. Include a method named get_color to return the vehicle's color. Print out the color of the vehicle.
7. Construct a new class named Car based on the Vehicle class, and override the get_colormethod so it always returns blue. Print out the color of the car.
8. Create a Car class based on two modules, one that contains a get_color method, and one that contains a get_number_of_wheels method. Print out the color of the car, and the number of wheels.

9.Create a web application named test with a controller named do and an action named greeting that displays the text “Hello”.
10. Modify the test application to store its “Hello” message in the greeting action.
11. Add a second action named greeting2 that displays “Hello again” and link to it from the greeting action’s view template. Add a second text field to the textfields example (which uses <input> elements to create a text field) to get the user’s age, and then display that age.
12. Add a second text field to the textfields2 example (which uses the text_field_tag method to create a text field) to get the user’s age, and then display that age.
13. Add a second text field to the textfields3 example (which uses the text_field method to create a text field) to get the user’s age, and then display that age.
14. Configure a Rails application to connect to a database server with the username orson_welles and the password rosebud.
15. Use the scaffold utility to create a model named item and a controller named merchandise.
16. If you know CSS, set the font size of scaffold-generated views to 16 points.
17. Add a text field and a Submit button to the display cart view asking for the user’s name so that he can check out.
18. Add a checkout action to the buy controller that recovers the customer’s name and the amount he owes.
19. Add a checkout view that displays the user’s name and how much he owes to let him check out.