Web Technologies
UNIT-VI

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.
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 Ruby 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

For performing the examples discussed in this tutorial, 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

This tutorial will provide the necessary skills to create GUI, networking, and Web applications using Ruby. It also will talk about extending and embedding Ruby applications.

Popular Ruby Editors:

To write your Ruby programs, you will need an editor:

- If you are working on Windows machine, then you can use any simple text editor like Notepad or Edit plus.
- VIM (Vi IMproved) is 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 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>
```

Ruby Syntax:

Let us write a simple program in ruby. All ruby files will have extension **.rb**. So, put the following source code in a test.rb file.

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

puts "Hello, Ruby!";
```

Here, I assumed that you have Ruby interpreter available in /usr/bin directory. Now, try to run this program as follows:

```
$ ruby test.rb
```

This will produce the following result:

```
Hello, Ruby!
```

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 mean Ram and RAM are two different idendifiers in Ruby.

Ruby identifier names may consist of alphanumeric characters and the underscore character (_).

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.

BEGIN do next then

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

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.
Here are different examples:

```
#!/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.
```

Ruby *BEGIN* Statement

Syntax:

```
BEGIN {
   code
}
```

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

Example:

```
#!/usr/bin/ruby

puts "This is main Ruby Program"

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

This will produce the following result:

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

Ruby *END* Statement

Syntax:

```
END {
   code
}
```

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

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
```

Ruby Comments:

A comment hides a line, part of a line, or several lines from the Ruby interpreter. You can use the hash character (#) at the beginning of a line:

```
# I am a comment. Just ignore me.
```

Or, a comment may be on the same line after a statement or expression:

```
name = "Madisetti" # This is again comment
```

You can comment multiple lines as follows:

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Here is another form. This block comment conceals several lines from the interpreter with =begin/=end:

```
=begin
This is a comment.
This is a comment, too.
This is a comment, too.
I said that already.
=end
```

Ruby Classes:

Ruby is a perfect Object Oriented Programming Language. The features of the object-oriented programming language include:

- Data Encapsulation:
- Data Abstraction:

- Polymorphism:
- Inheritance:

An object-oriented program involves classes and objects. A class is the blueprint from which individual objects are created. In object-oriented terms, we say that your *bicycle* is an instance of the *class of objects* known as bicycles.

Take the example of any vehicle. It comprises wheels, horsepower, and fuel or gas tank capacity. These characteristics form the data members of the class Vehicle. You can differentiate one vehicle from the other with the help of these characteristics.

A vehicle can also have certain functions, such as halting, driving, and speeding. Even these functions form the data members of the class Vehicle. You can, therefore, define a class as a combination of characteristics and functions.

A class Vehicle can be defined in Java as follows :

```
Class Vehicle
{
   Number no_of_wheels
   Number horsepower
   Characters type_of_tank
   Number Capacity
   Function speeding
   {
   }
   Function driving
   {
   }
   Function halting
   {
   }
}
```

By assigning different values to these data members, you can form several instances of the class Vehicle. For example, an airplane has three wheels, horsepower of 1,000, fuel as the type of tank, and a capacity of 100 liters. In the same way, a car has four wheels, horsepower of 200, gas as the type of tank, and a capacity of 25 litres.

Defining a Class in Ruby:

To implement object-oriented programming by using Ruby, you need to first learn how to create objects and classes in Ruby.

A class in Ruby always starts with the keyword *class* followed by the name of the class. The name should always be in initial capitals. The class *Customer* can be displayed as:

```
class Customer
end
```

You terminate a class by using the keyword *end*. All the data members in the *class* are between the class definition and the *end* keyword.

Variables in a Ruby Class:

Ruby provides four types of variables:

- **Local Variables:** Local variables are the variables that are defined in a method. Local variables are not available outside the method. You will see more details about method in subsequent chapter. Local variables begin with a lowercase letter or _.
- **Instance Variables:** Instance variables are available across methods for any particular instance or object. That means that instance variables change from object to object. Instance variables are preceded by the at sign (@) followed by the variable name.
- **Class Variables:** Class variables are available across different objects. A class variable belongs to the class and is a characteristic of a class. They are preceded by the sign @@ and are followed by the variable name.

- **Global Variables:** Class variables are not available across classes. If you want to have a single variable, which is available across classes, you need to define a global variable. The global variables are always preceded by the dollar sign ($).

Example:

Using the class variable @@no_of_customers, you can determine the number of objects that are being created. This enables in deriving the number of customers.

```
class Customer
   @@no_of_customers=0
end
```

Creating Objects in Ruby using *new* Method:

Objects are instances of the class. You will now learn how to create objects of a class in Ruby. You can create objects in Ruby by using the method *new* of the class.

The method *new* is a unique type of method, which is predefined in the Ruby library. The new method belongs to the *class* methods.

Here is the example to create two objects cust1 and cust2 of the class Customer:

```
cust1 = Customer. new
cust2 = Customer. new
```

Here, cust1 and cust2 are the names of two objects. You write the object name followed by the equal to sign (=) after which the class name will follow. Then, the dot operator and the keyword *new* will follow.

Custom Method to create Ruby Objects :

You can pass parameters to method *new* and those parameters can be used to initialize class variables.

When you plan to declare the *new* method with parameters, you need to declare the method *initialize* at the time of the class creation.

The *initialize* method is a special type of method, which will be executed when the *new* method of the class is called with parameters.

Here is the example to create initialize method:

```
class Customer
   @@no_of_customers=0
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
end
```

In this example, you declare the *initialize* method with **id, name**, and **addr** as local variables. Here, *def* and *end* are used to define a Ruby method *initialize*. You will learn more about methods in subsequent chapters.

In the *initialize* method, you pass on the values of these local variables to the instance variables @cust_id, @cust_name, and @cust_addr. Here local variables hold the values that are passed along with the new method.

Now, you can create objects as follows:

```
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")
```

Member Functions in Ruby Class:

In Ruby, functions are called methods. Each method in a *class* starts with the keyword *def* followed by the method name.

The method name always preferred in **lowercase letters**. You end a method in Ruby by using the keyword *end*.

Here is the example to define a Ruby method:

```
class Sample
   def function
      statement 1
```

```
      statement 2
   end
end
```
Here, *statement 1* and *statement 2* are part of the body of the method *function* inside the class Sample. These statments could be any valid Ruby statement. For example we can put a method *puts* to print *Hello Ruby* as follows:
```
class Sample
   def hello
      puts "Hello Ruby!"
   end
end
```
Now in the following example, create one object of Sample class and call *hello* method and see the result:
```
#!/usr/bin/ruby

class Sample
   def hello
      puts "Hello Ruby!"
   end
end

# Now using above class to create objects
object = Sample. new
object.hello
```
This will produce the following result:
```
Hello Ruby!
```
Ruby Variables

Variables are the memory locations which hold any data to be used by any program.
There are five types of variables supported by Ruby. You already have gone through a small description of these variables in previous chapter as well. These five types of variables are explained in this chapter.

Ruby Global Variables:

Global variables begin with $. Uninitialized global variables have the value *nil* and produce warnings with the -w option.
Assignment to global variables alters global status. It is not recommended to use global variables. They make programs cryptic.
Here is an example showing usage of global variable.
```
#!/usr/bin/ruby

$global_variable = 10
class Class1
  def print_global
     puts "Global variable in Class1 is #$global_variable"
  end
end
class Class2
  def print_global
     puts "Global variable in Class2 is #$global_variable"
  end
end

class1obj = Class1.new
class1obj.print_global
class2obj = Class2.new
class2obj.print_global
```
Here $global_variable is a global variable. This will produce the following result:

NOTE: In Ruby you CAN access value of any variable or constant by putting a hash (#) character just before that variable or constant.

```
Global variable in Class1 is 10
Global variable in Class2 is 10
```

Ruby Instance Variables:

Instance variables begin with @. Uninitialized instance variables have the value *nil* and produce warnings with the -w option.

Here is an example showing usage of Instance Variables.

```
#!/usr/bin/ruby

class Customer
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
    end
end

# Create Objects
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

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

Here, @cust_id, @cust_name and @cust_addr are instance variables. This will produce the following result:

```
Customer id 1
Customer name John
Customer address Wisdom Apartments, Ludhiya
Customer id 2
Customer name Poul
Customer address New Empire road, Khandala
```

Ruby Class Variables:

Class variables begin with @@ and must be initialized before they can be used in method definitions.

Referencing an uninitialized class variable produces an error. Class variables are shared among descendants of the class or module in which the class variables are defined.

Overriding class variables produce warnings with the -w option.

Here is an example showing usage of class variable:

```
#!/usr/bin/ruby

class Customer
   @@no_of_customers=0
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
      @@no_of_customers += 1
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
    end
```

```
    def total_no_of_customers()
       puts "Total number of customers: #@@no_of_customers"
    end
end

# Create Objects
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

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

Here @@no_of_customers is a class variable. This will produce the following result:

```
Total number of customers: 1
Total number of customers: 2
```

Ruby Local Variables:

Local variables begin with a lowercase letter or _. The scope of a local variable ranges from class, module, def, or do to the corresponding end or from a block's opening brace to its close brace {}.

When an uninitialized local variable is referenced, it is interpreted as a call to a method that has no arguments.

Assignment to uninitialized local variables also serves as variable declaration. The variables start to exist until the end of the current scope is reached. The lifetime of local variables is determined when Ruby parses the program.

In the above example local variables are id, name and addr.

Ruby Constants:

Constants begin with an uppercase letter. Constants defined within a class or module can be accessed from within that class or module, and those defined outside a class or module can be accessed globally.

Constants may not be defined within methods. Referencing an uninitialized constant produces an error. Making an assignment to a constant that is already initialized produces a warning.

```
#!/usr/bin/ruby

class Example
   VAR1 = 100
   VAR2 = 200
   def show
       puts "Value of first Constant is #{VAR1}"
       puts "Value of second Constant is #{VAR2}"
   end
end

# Create Objects
object=Example.new()
object.show
```

Here VAR1 and VAR2 are constant. This will produce the following result:

```
Value of first Constant is 100
Value of second Constant is 200
```

Ruby Pseudo-Variables:

They are special variables that have the appearance of local variables but behave like constants. You can not assign any value to these variables.

- **self:** The receiver object of the current method.
- **true:** Value representing true.
- **false:** Value representing false.
- **nil:** Value representing undefined.
- **__FILE__:** The name of the current source file.
- **__LINE__:** The current line number in the source file.

Ruby Basic Literals:

The rules Ruby uses for literals are simple and intuitive. This section explains all basic Ruby Literals.

Integer Numbers:

Ruby supports integer numbers. An integer number can range from -2^{30} to 2^{30-1} or -2^{62} to 2^{62-1}. Integers with-in this range are objects of class *Fixnum* and integers outside this range are stored in objects of class *Bignum*.

You write integers using an optional leading sign, an optional base indicator (0 for octal, 0x for hex, or 0b for binary), followed by a string of digits in the appropriate base. Underscore characters are ignored in the digit string.

You can also get the integer value corresponding to an ASCII character or escape sequence by preceding it with a question mark.

Example:

```
123                  # Fixnum decimal
1_234                # Fixnum decimal with underline
-500                 # Negative Fixnum
0377                 # octal
0xff                 # hexadecimal
0b1011               # binary
?a                   # character code for 'a'
?\n                  # code for a newline (0x0a)
12345678901234567890 # Bignum
```

NOTE: Class and Objects are explained in a separate chapter of this tutorial.

Floating Numbers:

Ruby supports integer numbers. They are also numbers but with decimals. Floating-point numbers are objects of class *Float* and can be any of the following:

Example:

```
123.4                # floating point value
1.0e6                # scientific notation
4E20                 # dot not required
4e+20                # sign before exponential
```

String Literals:

Ruby strings are simply sequences of 8-bit bytes and they are objects of class String. Double-quoted strings allow substitution and backslash notation but single-quoted strings don't allow substitution and allow backslash notation only for \\ and \'

Example:

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

puts 'escape using "\\"';
puts 'That\'s right';
```

This will produce the following result:

```
escape using "\"
That's right
```

You can substitute the value of any Ruby expression into a string using the sequence **#{ expr }**. Here, expr could be any ruby expression.

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

puts "Multiplication Value : #{24*60*60}";
```

This will produce the following result:

```
Multiplication Value : 86400
```

Backslash Notations:

Following is the list of Backslash notations supported by Ruby:

Notation	Character represented
\n	Newline (0x0a)

\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\a	Bell (0x07)
\e	Escape (0x1b)
\s	Space (0x20)
\nnn	Octal notation (n being 0-7)
\xnn	Hexadecimal notation (n being 0-9, a-f, or A-F)
\cx, \C-x	Control-x
\M-x	Meta-x (c | 0x80)
\M-\C-x	Meta-Control-x
\x	Character x

Ruby Ranges:

A Range represents an interval.a set of values with a start and an end. Ranges may be constructed using the s..e and s...e literals, or with Range.new.
Ranges constructed using .. run from the start to the end inclusively. Those created using ... exclude the end value. When used as an iterator, ranges return each value in the sequence.
A range (1..5) means it includes 1, 2, 3, 4, 5 values and a range (1...5) means it includes 1, 2, 3, 4 values.

Example:

```
#!/usr/bin/ruby

(10..15).each do |n|
   print n, ' '
end
```

This will produce the following result:

```
10 11 12 13 14 15
```

Ruby Conditional statements:

Ruby *if...else* Statement:

Syntax:

```
if conditional [then]
          code...
[elsif conditional [then]
          code...]...
[else
          code...]
end
```

if expressions are used for conditional execution. The values *false* and *nil* are false, and everything else are true. Notice Ruby uses elsif, not else if nor elif.
Executes *code* if the *conditional* is true. If the *conditional* is not true, *code* specified in the else clause is executed.
An if expression's *conditional* is separated from code by the reserved word *then*, a newline, or a semicolon.

Example:

```
#!/usr/bin/ruby

x=1
if x > 2
```

```
   puts "x is greater than 2"
elsif x <= 2 and x!=0
   puts "x is 1"
else
   puts "I can't guess the number"
end
x is 1
```

Ruby *if* modifier:

Syntax:

```
code if condition
```

Executes *code* if the *conditional* is true.

Example:

```
#!/usr/bin/ruby

$debug=1
print "debug\n" if $debug
```

This will produce the following result:

```
debug
```

Ruby *unless* Statement:

Syntax:

```
unless conditional [then]
   code
[else
   code ]
end
```

Executes *code* if *conditional* is false. If the *conditional* is true, code specified in the else clause is executed.

Example:

```
#!/usr/bin/ruby

x=1
unless x>2
   puts "x is less than 2"
 else
  puts "x is greater than 2"
end
```

This will produce the following result:

```
x is less than 2
```

Ruby *unless* modifier:

Syntax:

```
code unless conditional
```

Executes *code* if *conditional* is false.

Example:

```
#!/usr/bin/ruby

$var =  1
print "1 -- Value is set\n" if $var
print "2 -- Value is set\n" unless $var

$var = false
print "3 -- Value is set\n" unless $var
```

This will produce the following result:

```
1 -- Value is set
3 -- Value is set
```

Ruby *case* Statement

Syntax:

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

Compares the *expression* specified by case and that specified by when using the === operator and executes the *code* of the when clause that matches.

The *expression* specified by the when clause is evaluated as the left operand. If no when clauses match, *case* executes the code of the *else* clause.

A when statement's expression is separated from code by the reserved word then, a newline, or a semicolon.

Thus:

```
case expr0
when expr1, expr2
   stmt1
when expr3, expr4
   stmt2
else
   stmt3
end
```

is basically similar to the following:

```
_tmp = expr0
if expr1 === _tmp || expr2 === _tmp
   stmt1
elsif expr3 === _tmp || expr4 === _tmp
   stmt2
else
   stmt3
end
```

Example:

```
#!/usr/bin/ruby

$age =  5
case $age
when 0 .. 2
    puts "baby"
when 3 .. 6
    puts "little child"
when 7 .. 12
    puts "child"
when 13 .. 18
    puts "youth"
else
    puts "adult"
end
```

This will produce the following result:

```
little child
```

Ruby Looping statements:

Loops in Ruby are used to execute the same block of code a specified number of times. This chapter details all the loop statements supported by Ruby.

Ruby *while* Statement:

Syntax:

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

Executes *code* while *conditional* is true. A *while* loop's *conditional* is separated from *code* by the reserved word do, a newline, backslash \, or a semicolon ;.

Example:

```
#!/usr/bin/ruby

$i = 0
$num = 5

while $i < $num  do
   puts("Inside the loop i = #$i" )
   $i +=1
end
```

This will produce the following result:

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
```

Ruby *while* modifier:

Syntax:

```
code while condition

OR

begin
  code
end while conditional
```

Executes *code* while *conditional* is true.

If a *while* modifier follows a *begin* statement with no *rescue* or ensure clauses, *code* is executed once before conditional is evaluated.

Example:

```
#!/usr/bin/ruby

$i = 0
$num = 5
begin
   puts("Inside the loop i = #$i" )
   $i +=1
end while $i < $num
```

This will produce the following result:

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
```

Ruby *until* Statement:

```
until conditional [do]
   code
end
```

Executes *code* while *conditional* is false. An *until* statement's conditional is separated from *code* by the reserved word *do*, a newline, or a semicolon.

Example:

```
#!/usr/bin/ruby

$i = 0
$num = 5

until $i > $num  do
   puts("Inside the loop i = #$i" )
   $i +=1;
end
```

This will produce the following result:

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
Inside the loop i = 5
```

Ruby *until* modifier:

Syntax:

```
code until conditional

OR

begin
   code
end until conditional
```

Executes *code* while *conditional* is false.

If an *until* modifier follows a *begin* statement with no *rescue* or ensure clauses, *code* is executed once before *conditional* is evaluated.

Example:

```
#!/usr/bin/ruby

$i = 0
$num = 5
begin
   puts("Inside the loop i = #$i" )
   $i +=1;
end until $i > $num
```

This will produce the following result:

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
Inside the loop i = 5
```

Ruby *for* Statement:

Syntax:

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

Executes *code* once for each element in *expression*.

Example:

```
#!/usr/bin/ruby

for i in 0..5
   puts "Value of local variable is #{i}"
end
```

Here, we have defined the range 0..5. The statement for i in 0..5 will allow i to take values in the range from 0 to 5 (including 5). This will produce the following result:

```
Value of local variable is 0
Value of local variable is 1
Value of local variable is 2
Value of local variable is 3
Value of local variable is 4
Value of local variable is 5
```

A *for...in* loop is almost exactly equivalent to:

```
(expression).each do |variable[, variable...]| code end
```

except that a for loop doesn't create a new scope for local variables. A for loop's *expression* is separated from *code* by the reserved word do, a newline, or a semicolon.

Example:

```
#!/usr/bin/ruby

(0..5).each do |i|
   puts "Value of local variable is #{i}"
end
```

This will produce the following result:

```
Value of local variable is 0
Value of local variable is 1
Value of local variable is 2
Value of local variable is 3
Value of local variable is 4
Value of local variable is 5
```

Ruby *break* Statement:

Syntax:

```
break
```

Terminates the most internal loop. Terminates a method with an associated block if called within the block (with the method returning nil).

Example:

```
#!/usr/bin/ruby

for i in 0..5
   if i > 2 then
      break
   end
   puts "Value of local variable is #{i}"
end
```

This will produce the following result:

```
Value of local variable is 0
Value of local variable is 1
Value of local variable is 2
```

Ruby *next* Statement:

Syntax:

```
next
```

Jumps to next iteration of the most internal loop. Terminates execution of a block if called within a block (with *yield* or call returning nil).

Example:

```
#!/usr/bin/ruby

for i in 0..5
   if i < 2 then
      next
   end
   puts "Value of local variable is #{i}"
end
```

This will produce the following result:

```
Value of local variable is 2
Value of local variable is 3
Value of local variable is 4
Value of local variable is 5
```

Ruby *redo* Statement:

Syntax:

```
redo
```

Restarts this iteration of the most internal loop, without checking loop condition. Restarts *yield* or *call* if called within a block.

Example:

```
#!/usr/bin/ruby

for i in 0..5
   if i < 2 then
      puts "Value of local variable is #{i}"
      redo
   end
end
```

This will produce the following result and will go in an infinite loop:

```
Value of local variable is 0
Value of local variable is 0
.............................
```

Ruby *retry* Statement:

Syntax:

```
retry
```

If *retry* appears in rescue clause of begin expression, restart from the beginning of the 1begin body.

```
begin
   do_something # exception raised
rescue
   # handles error
   retry  # restart from beginning
end
```

If retry appears in the iterator, the block, or the body of the for expression, restarts the invocation of the iterator call. Arguments to the iterator is re-evaluated.

```
for i in 1..5
   retry if some_condition # restart from i == 1
end
```

Example:

```
#!/usr/bin/ruby

for i in 1..5
   retry if  i > 2
   puts "Value of local variable is #{i}"
end
```

This will produce the following result and will go in an infinite loop:

```
Value of local variable is 1
Value of local variable is 2
Value of local variable is 1
Value of local variable is 2
Value of local variable is 1
Value of local variable is 2
.............................
```

Ruby Methods

Ruby methods are very similar to functions in any other programming language. Ruby methods are used to bundle one or more repeatable statements into a single unit.
Method names should begin with a lowercase letter. If you begin a method name with an uppercase letter, Ruby might think that it is a constant and hence can parse the call incorrectly.
Methods should be defined before calling them, otherwise Ruby will raise an exception for undefined method invoking.

Syntax:

```
def method_name [( [arg [= default]]...[, * arg [, &expr ]])]
   expr..
end
```

So you can define a simple method as follows:

```
def method_name
   expr..
end
```

You can represent a method that accepts parameters like this:

```
def method_name (var1, var2)
   expr..
end
```

You can set default values for the parameters which will be used if method is called without passing required parameters:

```
def method_name (var1=value1, var2=value2)
   expr..
end
```

Whenever you call the simple method, you write only the method name as follows:

```
method_name
```

However, when you call a method with parameters, you write the method name along with the parameters, such as:

```
method_name 25, 30
```

The most important drawback to using methods with parameters is that you need to remember the number of parameters whenever you call such methods. For example, if a method accepts three parameters and you pass only two, then Ruby displays an error.

Example:

```
#!/usr/bin/ruby

def test(a1="Ruby", a2="Perl")
   puts "The programming language is #{a1}"
   puts "The programming language is #{a2}"
end
test "C", "C++"
test
```

This will produce the following result:

```
The programming language is C
The programming language is C++
The programming language is Ruby
The programming language is Perl
```

Return Values from Methods:

Every method in Ruby returns a value by default. This returned value will be the value of the last statement. For example:

```
def test
   i = 100
   j = 10
```

```
   k = 0
end
```
This method, when called, will return the last declared variable k.

Ruby *return* Statement:

The *return* statement in ruby is used to return one or more values from a Ruby Method.

Syntax:

```
return [expr[`,' expr...]]
```
If more than two expressions are given, the array containing these values will be the return value. If no expression given, nil will be the return value.

Example:

```
return

OR

return 12

OR

return 1,2,3
```
Have a look at this example:
```
#!/usr/bin/ruby

def test
   i = 100
   j = 200
   k = 300
return i, j, k
end
var = test
puts var
```
This will produce the following result:
```
100
200
300
```

Class Methods:

When a method is defined outside of the class definition, the method is marked as *private* by default. On the other hand, the methods defined in the class definition are marked as public by default. The default visibility and the *private* mark of the methods can be changed by *public* or *private* of the Module.

Whenever you want to access a method of a class, you first need to instantiate the class. Then, using the object, you can access any member of the class.

Ruby gives you a way to access a method without instantiating a class. Let us see how a class method is declared and accessed:
```
class Accounts
   def reading_charge
   end
   def Accounts.return_date
   end
end
```
See how the method return_date is declared. It is declared with the class name followed by a period, which is followed by the name of the method. You can access this class method directly as follows:
```
Accounts.return_date
```
To access this method, you need not create objects of the class Accounts.

Ruby *alias* Statement:

This gives alias to methods or global variables. Aliases can not be defined within the method body. The alias of the method keep the current definition of the method, even when methods are overridden.
Making aliases for the numbered global variables ($1, $2,...) is prohibited. Overriding the built-in global variables may cause serious problems.

Syntax:

```
alias method-name method-name
alias global-variable-name global-variable-name
```

Example:

```
alias foo bar
alias $MATCH $&
```

Here we have defined foo alias for bar and $MATCH is an alias for $&

Ruby *undef* Statement:

This cancels the method definition. An *undef* can not appear in the method body.
By using *undef* and *alias*, the interface of the class can be modified independently from the superclass, but notice it may be broke programs by the internal method call to self.

Syntax:

```
undef method-name
```

Example:

To undefine a method called *bar* do the following:

```
undef bar
```

Ruby Arrays

Ruby arrays are ordered, integer-indexed collections of any object. Each element in an array is associated with and referred to by an index.
Array indexing starts at 0, as in C or Java. A negative index is assumed relative to the end of the array --- that is, an index of -1 indicates the last element of the array, -2 is the next to last element in the array, and so on.
Ruby arrays can hold objects such as String, Integer, Fixnum, Hash, Symbol, even other Array objects. Ruby arrays are not as rigid as arrays in other languages. Ruby arrays grow automatically while adding elements to them.

Creating Arrays:

There are many ways to create or initialize an array. One way is with the *new* class method:

```
names = Array.new
```

You can set the size of an array at the time of creating array:

```
names = Array.new(20)
```

The array *names* now has a size or length of 20 elements. You can return the size of an array with either the size or length methods:

```
#!/usr/bin/ruby

names = Array.new(20)
puts names.size  # This returns 20
puts names.length # This also returns 20
```

This will produce the following result:

```
20
20
```

You can assign a value to each element in the array as follows:

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

```
names = Array.new(4, "mac")

puts "#{names}"
```
This will produce the following result:
```
macmacmacmac
```
You can also use a block with new, populating each element with what the block evaluates to:
```
#!/usr/bin/ruby

nums = Array.new(10) { |e| e = e * 2 }

puts "#{nums}"
```
This will produce the following result:
```
024681012141618
```
There is another method of Array, []. It works like this:
```
nums = Array.[](1, 2, 3, 4,5)
```
One more form of array creation is as follows :
```
nums = Array[1, 2, 3, 4,5]
```
The *Kernel* module available in core Ruby has an Array method, which only accepts a single argument. Here, the method takes a range as an argument to create an array of digits:
```
#!/usr/bin/ruby

digits = Array(0..9)

puts "#{digits}"
```
This will produce the following result:
```
0123456789
```

Ruby Hashes

A Hash is a collection of key-value pairs like this: "employee" => "salary". It is similar to an Array, except that indexing is done via arbitrary keys of any object type, not an integer index. The order in which you traverse a hash by either key or value may seem arbitrary and will generally not be in the insertion order. If you attempt to access a hash with a key that does not exist, the method will return *nil*.

Creating Hashes:

As with arrays, there is a variety of ways to create hashes. You can create an empty hash with the *new* class method:
```
months = Hash.new
```
You can also use *new* to create a hash with a default value, which is otherwise just *nil*:
```
months = Hash.new( "month" )

or

months = Hash.new "month"
```
When you access any key in a hash that has a default value, if the key or value doesn't exist, accessing the hash will return the default value:
```
#!/usr/bin/ruby

months = Hash.new( "month" )

puts "#{months[0]}"
puts "#{months[72]}"
```
This will produce the following result:
```
month
month
#!/usr/bin/ruby
```

```
H = Hash["a" => 100, "b" => 200]

puts "#{H['a']}"
puts "#{H['b']}"
```
This will produce the following result:
```
100
200
```
You can use any Ruby object as a key or value, even an array, so following example is a valid one:
```
[1,"jan"] => "January"
```

Ruby Iterators - each and collect

Iterators are nothing but methods supported by *collections*. Objects that store a group of data members are called collections. In Ruby, arrays and hashes can be termed collections. Iterators return all the elements of a collection, one after the other. We will be discussing two iterators here, *each* and *collect*. Let's look at these in detail.

Ruby *each* Iterator:

The each iterator returns all the elements of an array or a hash.

Syntax:

```
collection.each do |variable|
   code
end
```
Executes *code* for each element in *collection*. Here, *collection* could be an array or a ruby hash.

Example:

```
#!/usr/bin/ruby

ary = [1,2,3,4,5]
ary.each do |i|
   puts i
end
```
This will produce the following result:
```
1
2
3
4
5
```
You always associate the *each* iterator with a block. It returns each value of the array, one by one, to the block. The value is stored in the variable **i** and then displayed on the screen.

Ruby *collect* Iterator:

The *collect* iterator returns all the elements of a collection.

Syntax:

```
collection = collection.collect
```
The *collect* method need not always be associated with a block. The *collect* method returns the entire collection, regardless of whether it is an array or a hash.

Example:

```
#!/usr/bin/ruby

a = [1,2,3,4,5]
b = Array.new
b = a.collect{ |e| e }
```

```
puts b
```
This will produce the following result:
```
1
2
3
4
5
```
NOTE: The *collect* method is not the right way to do copying between arrays. There is another method called a *clone*, which should be used to copy one array into another array. You normally use the collect method when you want to do something with each of the values to get the new array. For example, this code produces an array b containing 10 times each value in a.
```
#!/usr/bin/ruby

a = [1,2,3,4,5]
b = a.collect{|x| 10*x}
puts b
```
This will produce the following result:
```
10
20
30
40
50
```
Ruby File I/O, Directories
Ruby provides a whole set of I/O-related methods implemented in the Kernel module. All the I/O methods are derived from the class IO.
The class *IO* provides all the basic methods, such as *read*, *write*, *gets*, *puts*, *readline*, *getc*, and *printf*.
This chapter will cover all the basic I/O functions available in Ruby. For more functions, please refer to Ruby Class *IO*.
The *puts* Statement:
In previous chapters, you assigned values to variables and then printed the output using *puts* statement.
The *puts* statement instructs the program to display the value stored in the variable. This will add a new line at the end of each line it writes.
Example:
```
#!/usr/bin/ruby

val1 = "This is variable one"
val2 = "This is variable two"
puts val1
puts val2
```
This will produce the following result:
```
This is variable one
This is variable two
```
The *gets* Statement:
The *gets* statement can be used to take any input from the user from standard screen called STDIN.
Example:
The following code shows you how to use the gets statement. This code will prompt the user to enter a value, which will be stored in a variable val and finally will be printed on STDOUT.
```
#!/usr/bin/ruby

puts "Enter a value :"
val = gets
```

```
puts val
```
This will produce the following result:
```
Enter a value :
This is entered value
This is entered value
```
The *putc* Statement:
Unlike the *puts* statement, which outputs the entire string onto the screen, the *putc* statement can be used to output one character at a time.
Example:
The output of the following code is just the character H:
```
#!/usr/bin/ruby

str="Hello Ruby!"
putc str
```
This will produce the following result:
```
H
```
The *print* Statement:
The *print* statement is similar to the *puts* statement. The only difference is that the *puts* statement goes to the next line after printing the contents, whereas with the *print* statement the cursor is positioned on the same line.
Example:
```
#!/usr/bin/ruby

print "Hello World"
print "Good Morning"
```
This will produce the following result:
```
Hello WorldGood Morning
```
Opening and Closing Files:
Until now, you have been reading and writing to the standard input and output. Now, we will see how to play with actual data files.
The *File.new* Method:
You can create a *File* object using *File.new* method for reading, writing, or both, according to the mode string. Finally, you can use *File.close* method to close that file.
Syntax:
```
aFile = File.new("filename", "mode")
   # ... process the file
aFile.close
```
The *File.open* Method:
You can use *File.open* method to create a new file object and assign that file object to a file. However, there is one difference in between *File.open* and *File.new* methods. The difference is that the *File.open* method can be associated with a block, whereas you cannot do the same using the *File.new* method.
```
File.open("filename", "mode") do |aFile|
   # ... process the file
end
```

Here is a list of The Different Modes of Opening a File:

Modes	Description
r	Read-only mode. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Read-write mode. The file pointer will be at the beginning of the file.

w	Write-only mode. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Read-write mode. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Write-only mode. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Read and write mode. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

Reading and Writing Files:

The same methods that we've been using for 'simple' I/O are available for all file objects. So, gets reads a line from standard input, and *aFile.gets* reads a line from the file object aFile. However, I/O objects provides additional set of access methods to make our lives easier.

The *sysread* Method:

You can use the method *sysread* to read the contents of a file. You can open the file in any of the modes when using the method sysread. For example :

Following is the input text file:

```
This is a simple text file for testing purpose.
```

Now let's try to read this file:

```
#!/usr/bin/ruby

aFile = File.new("input.txt", "r")
if aFile
   content = aFile.sysread(20)
   puts content
else
   puts "Unable to open file!"
end
```

This statement will output the first 20 characters of the file. The file pointer will now be placed at the 21st character in the file.

The *syswrite* Method:

You can use the method syswrite to write the contents into a file. You need to open the file in write mode when using the method syswrite. For example:

```
#!/usr/bin/ruby

aFile = File.new("input.txt", "r+")
if aFile
   aFile.syswrite("ABCDEF")
else
   puts "Unable to open file!"
end
```

This statement will write "ABCDEF" into the file.

The *each_byte* Method:

This method belongs to the class *File*. The method *each_byte* is always associated with a block. Consider the following code sample:

```
#!/usr/bin/ruby

aFile = File.new("input.txt", "r+")
if aFile
   aFile.syswrite("ABCDEF")
   aFile.each_byte {|ch| putc ch; putc ?. }
else
   puts "Unable to open file!"
end
```

Characters are passed one by one to the variable ch and then displayed on the screen as follows:

```
s. .a. .s.i.m.p.l.e. .t.e.x.t. .f.i.l.e. .f.o.r. .t.e.s.t.i.n.g.
.p.u.r.p.o.s.e...
.
.
```

The *IO.readlines* Method:

The class *File* is a subclass of the class IO. The class IO also has some methods, which can be used to manipulate files.

One of the IO class methods is *IO.readlines*. This method returns the contents of the file line by line. The following code displays the use of the method *IO.readlines*:

```
#!/usr/bin/ruby

arr = IO.readlines("input.txt")
puts arr[0]
puts arr[1]
```

In this code, the variable arr is an array. Each line of the file *input.txt* will be an element in the array arr. Therefore, arr[0] will contain the first line, whereas arr[1] will contain the second line of the file.

The *IO.foreach* Method:

This method also returns output line by line. The difference between the method *foreach* and the method *readlines* is that the method *foreach* is associated with a block. However, unlike the method *readlines*, the method *foreach* does not return an array. For example:

```
#!/usr/bin/ruby

IO.foreach("input.txt"){|block| puts block}
```

This code will pass the contents of the file *test* line by line to the variable block, and then the output will be displayed on the screen.

Renaming and Deleting Files:

You can rename and delete files programmatically with Ruby with the *rename* and *delete* methods.

Following is the example to rename an existing file *test1.txt*:

```
#!/usr/bin/ruby

# Rename a file from test1.txt to test2.txt
File.rename( "test1.txt", "test2.txt" )
```

Following is the example to delete an existing file *test2.txt*:

```
#!/usr/bin/ruby

# Delete file test2.txt
File.delete("test2.txt")
```

File Modes and Ownership:

Use the *chmod* method with a mask to change the mode or permissions/access list of a file:

Following is the example to change mode of an existing file *test.txt* to a mask value:

```
#!/usr/bin/ruby

file = File.new( "test.txt", "w" )
file.chmod( 0755 )
```

Following is the table, which can help you to choose different mask for *chmod* method:

Mask	Description
0700	rwx mask for owner
0400	r for owner
0200	w for owner
0100	x for owner
0070	rwx mask for group
0040	r for group

0020	w for group
0010	x for group
0007	rwx mask for other
0004	r for other
0002	w for other
0001	x for other
4000	Set user ID on execution
2000	Set group ID on execution
1000	Save swapped text, even after use

File Inquiries:

The following command tests whether a file exists before opening it:

```
#!/usr/bin/ruby

File.open("file.rb") if File::exists?( "file.rb" )
```

The following command inquire whether the file is really a file:

```
#!/usr/bin/ruby

# This returns either true or false
File.file?( "text.txt" )
```

The following command finds out if it given file name is a directory:

```
#!/usr/bin/ruby

# a directory
File::directory?( "/usr/local/bin" ) # => true

# a file
File::directory?( "file.rb" ) # => false
```

The following command finds whether the file is readable, writable or executable:

```
#!/usr/bin/ruby

File.readable?( "test.txt" )   # => true
File.writable?( "test.txt" )   # => true
File.executable?( "test.txt" ) # => false
```

The following command finds whether the file has zero size or not:

```
#!/usr/bin/ruby

File.zero?( "test.txt" )      # => true
```

The following command returns size of the file :

```
#!/usr/bin/ruby

File.size?( "text.txt" )     # => 1002
```

The following command can be used to find out a type of file :

```
#!/usr/bin/ruby

File::ftype( "test.txt" )      # => file
```

The ftype method identifies the type of the file by returning one of the following: *file, directory, characterSpecial, blockSpecial, fifo, link, socket, or unknown.*

The following command can be used to find when a file was created, modified, or last accessed :

```
#!/usr/bin/ruby

File::ctime( "test.txt" ) # => Fri May 09 10:06:37 -0700 2008
File::mtime( "text.txt" ) # => Fri May 09 10:44:44 -0700 2008
File::atime( "text.txt" ) # => Fri May 09 10:45:01 -0700 2008
```

Directories in Ruby:

All files are contained within various directories, and Ruby has no problem handling these too. Whereas the *File* class handles files, directories are handled with the *Dir* class.

Navigating Through Directories:

To change directory within a Ruby program, use *Dir.chdir* as follows. This example changes the current directory to */usr/bin*.

```
Dir.chdir("/usr/bin")
```

You can find out what the current directory is with *Dir.pwd*:

```
puts Dir.pwd # This will return something like /usr/bin
```

You can get a list of the files and directories within a specific directory using *Dir.entries*:

```
puts Dir.entries("/usr/bin").join(' ')
```

Dir.entries returns an array with all the entries within the specified directory. *Dir.foreach* provides the same feature:

```
Dir.foreach("/usr/bin") do |entry|
   puts entry
end
```

An even more concise way of getting directory listings is by using Dir's class array method:

```
Dir["/usr/bin/*"]
```

Creating a Directory:

The *Dir.mkdir* can be used to create directories:

```
Dir.mkdir("mynewdir")
```

You can also set permissions on a new directory (not one that already exists) with mkdir:

NOTE: The mask 755 sets permissions owner, group, world [anyone] to rwxr-xr-x where r = read, w = write, and x = execute.

```
Dir.mkdir( "mynewdir", 755 )
```

Deleting a Directory:

The *Dir.delete* can be used to delete a directory. The *Dir.unlink* and *Dir.rmdir* perform exactly the same function and are provided for convenience.

```
Dir.delete("testdir")
```

Creating Files & Temporary Directories:

Temporary files are those that might be created briefly during a program's execution but aren't a permanent store of information.

Dir.tmpdir provides the path to the temporary directory on the current system, although the method is not available by default. To make *Dir.tmpdir* available it's necessary to use require 'tmpdir'.

You can use *Dir.tmpdir* with *File.join* to create a platform-independent temporary file:

```
require 'tmpdir'
   tempfilename = File.join(Dir.tmpdir, "tingtong")
   tempfile = File.new(tempfilename, "w")
   tempfile.puts "This is a temporary file"
   tempfile.close
   File.delete(tempfilename)
```

This code creates a temporary file, writes data to it, and deletes it. Ruby's standard library also includes a library called *Tempfile* that can create temporary files for you:

```
require 'tempfile'
   f = Tempfile.new('tingtong')
   f.puts "Hello"
   puts f.path
   f.close
```

Ruby Regular Expressions

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings using a specialized syntax held in a pattern.
A *regular expression literal* is a pattern between slashes or between arbitrary delimiters followed by %r as follows:

Syntax:

```
/pattern/
/pattern/im    # option can be specified
%r!/usr/local! # general delimited regular expression
```

Example:

```
#!/usr/bin/ruby

line1 = "Cats are smarter than dogs";
line2 = "Dogs also like meat";

if ( line1 =~ /Cats(.*)/ )
  puts "Line1 contains Cats"
end
if ( line2 =~ /Cats(.*)/ )
  puts "Line2 contains  Dogs"
end
```

This will produce the following result:

```
Line1 contains Cats
```

Regular-expression modifiers:

Regular expression literals may include an optional modifier to control various aspects of matching. The modifier is specified after the second slash character, as shown previously and may be represented by one of these characters:

Modifier	Description
i	Ignore case when matching text.
o	Perform #{} interpolations only once, the first time the regexp literal is evaluated.
x	Ignores whitespace and allows comments in regular expressions
m	Matches multiple lines, recognizing newlines as normal characters
u,e,s,n	Interpret the regexp as Unicode (UTF-8), EUC, SJIS, or ASCII. If none of these modifiers is specified, the regular expression is assumed to use the source encoding.

Like string literals delimited with %Q, Ruby allows you to begin your regular expressions with %r followed by a delimiter of your choice. This is useful when the pattern you are describing contains a lot of forward slash characters that you don't want to escape:

```
# Following matches a single slash character, no escape required
%r|/|

# Flag characters are allowed with this syntax, too
%r[</(.*)>]i
```

Regular-expression patterns:

Except for control characters, **(+ ? . * ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.
Following table lists the regular expression syntax that is available in Ruby.

Pattern	Description

^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more occurrence of preceding expression.
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers matched text.
(?imx)	Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?-imx)	Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?: re)	Groups regular expressions without remembering matched text.
(?imx: re)	Temporarily toggles on i, m, or x options within parentheses.
(?-imx: re)	Temporarily toggles off i, m, or x options within parentheses.
(?#...)	Comment.
(?= re)	Specifies position using a pattern. Doesn't have a range.
(?! re)	Specifies position using pattern negation. Doesn't have a range.
(?> re)	Matches independent pattern without backtracking.
\w	Matches word characters.
\W	Matches nonword characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\G	Matches point where last match finished.
\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\B	Matches nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\1...\9	Matches nth grouped subexpression.
\10	Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

Regular-expression Examples:
Literal characters:

Example	Description
/ruby/	Match "ruby".
¥	Matches Yen sign. Multibyte characters are supported in Ruby 1.9 and Ruby 1.8.

Character classes:

Example	Description
/[Rr]uby/	Match "Ruby" or "ruby"
/rub[ye]/	Match "ruby" or "rube"
/[aeiou]/	Match any one lowercase vowel
/[0-9]/	Match any digit; same as /[0123456789]/
/[a-z]/	Match any lowercase ASCII letter
/[A-Z]/	Match any uppercase ASCII letter
/[a-zA-Z0-9]/	Match any of the above
/[^aeiou]/	Match anything other than a lowercase vowel
/[^0-9]/	Match anything other than a digit

Special Character Classes:

Example	Description
/./	Match any character except newline
/./m	In multiline mode . matches newline, too
/\d/	Match a digit: /[0-9]/
/\D/	Match a nondigit: /[^0-9]/
/\s/	Match a whitespace character: /[\t\r\n\f]/
/\S/	Match nonwhitespace: /[^ \t\r\n\f]/
/\w/	Match a single word character: /[A-Za-z0-9_]/
/\W/	Match a nonword character: /[^A-Za-z0-9_]/

Repetition Cases:

Example	Description
/ruby?/	Match "rub" or "ruby": the y is optional
/ruby*/	Match "rub" plus 0 or more ys
/ruby+/	Match "rub" plus 1 or more ys
/\d{3}/	Match exactly 3 digits
/\d{3,}/	Match 3 or more digits
/\d{3,5}/	Match 3, 4, or 5 digits

Nongreedy repetition:

This matches the smallest number of repetitions:

Example	Description
/<.*>/	Greedy repetition: matches "<ruby>perl>"
/<.*?>/	Nongreedy: matches "<ruby>" in "<ruby>perl>"

Grouping with parentheses:

Example	Description

/\D\d+/	No group: + repeats \d
/(\D\d)+/	Grouped: + repeats \D\d pair
/([Rr]uby(,)?)+/	Match "Ruby", "Ruby, ruby, ruby", etc.

Backreferences:

This matches a previously matched group again:

Example	Description
/([Rr])uby&\1ails/	Match ruby&rails or Ruby&Rails
/(['"])(?:(?!\1).)*\1/	Single or double-quoted string. \1 matches whatever the 1st group matched . \2 matches whatever the 2nd group matched, etc.

Alternatives:

Example	Description
/ruby|rube/	Match "ruby" or "rube"
/rub(y|le))/	Match "ruby" or "ruble"
/ruby(!+|\?)/	"ruby" followed by one or more ! or one ?

Anchors:

This need to specify match position

Example	Description
/^Ruby/	Match "Ruby" at the start of a string or internal line
/Ruby$/	Match "Ruby" at the end of a string or line
/\ARuby/	Match "Ruby" at the start of a string
/Ruby\Z/	Match "Ruby" at the end of a string
/\bRuby\b/	Match "Ruby" at a word boundary
/\brub\B/	\B is nonword boundary: match "rub" in "rube" and "ruby" but not alone
/Ruby(?=!)/	Match "Ruby", if followed by an exclamation point
/Ruby(?!!)/	Match "Ruby", if not followed by an exclamation point

Special syntax with parentheses:

Example	Description
/R(?#comment)/	Matches "R". All the rest is a comment
/R(?i)uby/	Case-insensitive while matching "uby"
/R(?i:uby)/	Same as above
/rub(?:y|le))/	Group only without creating \1 backreference

Search and Replace:

Some of the most important String methods that use regular expressions are **sub** and **gsub** , and their in-place variants **sub!** and **gsub!**.

All of these methods perform a search-and-replace operation using a Regexp pattern. The **sub** & **sub!** replace the first occurrence of the pattern and **gsub** & **gsub!** replace all occurrences. The **sub** and **gsub** return a new string, leaving the original unmodified where as **sub!** and **gsub!** modify the string on which they are called.

Following is the example:

```
#!/usr/bin/ruby

phone = "2004-959-559 #This is Phone Number"

# Delete Ruby-style comments
phone = phone.sub!(/#.*$/, "")
puts "Phone Num : #{phone}"
```

```
# Remove anything other than digits
phone = phone.gsub!(/\D/, "")
puts "Phone Num : #{phone}"
```
This will produce the following result:
```
Phone Num : 2004-959-559
Phone Num : 2004959559
```
Following is another example:
```
#!/usr/bin/ruby

text = "rails are rails, really good Ruby on Rails"

# Change "rails" to "Rails" throughout
text.gsub!("rails", "Rails")

# Capitalize the word "Rails" throughout
text.gsub!(/\brails\b/, "Rails")

puts "#{text}"
```
This will produce the following result:
```
Rails are Rails, really good Ruby on Rails
```

Ruby Web Applications - CGI Programming

Ruby is a general-purpose language; it can't properly be called a *web language* at all. Even so, web applications and web tools in general are among the most common uses of Ruby. Not only can you write your own SMTP server, FTP daemon, or Web server in Ruby, but you can also use Ruby for more usual tasks such as CGI programming or as a replacement for PHP.
Please spend few minutes with CGI Programming Tutorial for more detail on CGI Programming.

Writing CGI Scripts:

The most basic Ruby CGI script looks like this:
```
#!/usr/bin/ruby

puts "HTTP/1.0 200 OK"
puts "Content-type: text/html\n\n"
puts "<html><body>This is a test</body></html>"
```
If you call this script *test.cgi* and uploaded it to a Unix-based Web hosting provider with the right permissions, you could use it as a CGI script.
For example, if you have the Web site http://www.example.com/ hosted with a Linux Web hosting provider and you upload *test.cgi* to the main directory and give it execute permissions, then visiting http://www.example.com/test.cgi should return an HTML page saying ***This is a test***.
Here when *test.cgi* is requested from a Web browser, the Web server looks for *test.cgi* on the Web site, and then executes it using the Ruby interpreter. The Ruby script returns a basic HTTP header and then returns a basic HTML document.

Using cgi.rb:

Ruby comes with a special library called **cgi** that enables more sophisticated interactions than those with the preceding CGI script.
Let's create a basic CGI script that uses cgi:
```
#!/usr/bin/ruby

require 'cgi'

cgi = CGI.new
puts cgi.header
puts "<html><body>This is a test</body></html>"
```

Here, you created a CGI object and used it to print the header line for you.

Form Processing:

Using class CGI gives you access to HTML query parameters in two ways. Suppose we are given a URL of /cgi-bin/test.cgi?FirstName=Zara&LastName=Ali.

You can access the parameters *FirstName* and *LastName* using CGI#[] directly as follows:

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
cgi['FirstName'] # =>  ["Zara"]
cgi['LastName']  # =>  ["Ali"]
```

There is another way to access these form variables. This code will give you a hash of all the key and values:

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
h = cgi.params  # =>  {"FirstName"=>["Zara"],"LastName"=>["Ali"]}
h['FirstName']  # =>  ["Zara"]
h['LastName']   # =>  ["Ali"]
```

Following is the code to retrieve all the keys:

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
cgi.keys         # =>  ["FirstName", "LastName"]
```

If a form contains multiple fields with the same name, the corresponding values will be returned to the script as an array. The [] accessor returns just the first of these.index the result of the params method to get them all.

In this example, assume the form has three fields called "name" and we entered three names "Zara", "Huma" and "Nuha":

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
cgi['name']         # => "Zara"
cgi.params['name'] # => ["Zara", "Huma", "Nuha"]
cgi.keys           # => ["name"]
cgi.params         # => {"name"=>["Zara", "Huma", "Nuha"]}
```

Note: Ruby will take care of GET and POST methods automatically. There is no separate treatment for these two different methods.

An associated, but basic, form that could send the correct data would have HTML code like so:

```
<html>
<body>
<form method="POST" action="http://www.example.com/test.cgi">
First Name :<input type="text" name="FirstName" value="" />
<br />
Last Name :<input type="text" name="LastName" value="" />

<input type="submit" value="Submit Data" />
</form>
</body>
</html>
```

Creating Forms and HTML:

CGI contains a huge number of methods used to create HTML. You will find one method per tag. In order to enable these methods, you must create a CGI object by calling CGI.new.

To make tag nesting easier, these methods take their content as code blocks. The code blocks should return a *String*, which will be used as the content for the tag. For example:

```
#!/usr/bin/ruby

require "cgi"
cgi = CGI.new("html4")
cgi.out{
   cgi.html{
      cgi.head{ "\n"+cgi.title{"This Is a Test"} } +
      cgi.body{ "\n"+
         cgi.form{"\n"+
            cgi.hr +
            cgi.h1 { "A Form: " } + "\n"+
            cgi.textarea("get_text") +"\n"+
            cgi.br +
            cgi.submit
         }
      }
   }
}
```

NOTE: The *form* method of the CGI class can accept a method parameter, which will set the HTTP method (GET, POST, and so on...) to be used on form submittal. The default, used in this example, is POST.

This will produce the following result:

```
Content-Type: text/html
Content-Length: 302
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Final//EN">
<HTML>
<HEAD>
<TITLE>This Is a Test</TITLE>
</HEAD>
<BODY>
<FORM METHOD="post" ENCTYPE="application/x-www-form-urlencoded">
<HR>
<H1>A Form: </H1>
<TEXTAREA COLS="70" NAME="get_text" ROWS="10"></TEXTAREA>
<BR>
<INPUT TYPE="submit">
</FORM>
</BODY>
</HTML>
```