


# Ruby - Quick Guide


Advertisements

 Previous Page

Next Page 

## Ruby - Overview

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](http://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 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.

## What is Next?

The next chapter guides you to where you can obtain Ruby and its documentation. Finally, it instructs you on how to install Ruby and prepare an environment to develop Ruby applications.

# Ruby - Environment Setup

## Local Environment Setup

If you are still willing to set up your environment for Ruby programming language, then let's proceed. This tutorial will teach you all the important topics related to environment setup. We would recommend you to go through the following topics first and then proceed further –

- ▣ [Ruby Installation on Linux/Unix](#) – If you are planning to have your development environment on Linux/Unix Machine, then go through this chapter.
- ▣ [Ruby Installation on Windows](#) – If you are planning to have your development environment on Windows Machine, then go through this chapter.
- ▣ [Ruby Command Line Options](#) – This chapter list out all the command line options, which you can use along with Ruby interpreter.
- ▣ [Ruby Environment Variables](#) – This chapter has a list of all the important environment variables to be set to make Ruby Interpreter works.

## 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 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>
```

Do not worry about what we did here. You will learn all these steps in subsequent chapters.

## What is Next?

We assume now you have a working Ruby Environment and you are ready to write the first Ruby Program. The next chapter will teach you how to write Ruby programs.

## 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!";
```

[Live Demo](#)

Here, we 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!
```

You have seen a simple Ruby program, now let us see a few basic concepts related to 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 ( \_ ).

## 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	while
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"
}
```

[Live Demo](#)

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

[Live Demo](#)

```
#!/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 and Objects

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

- ▣ Data Encapsulation
- ▣ Data Abstraction
- ▣ Polymorphism
- ▣ Inheritance

These features have been discussed in the chapter Object Oriented Ruby [↗](#).

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 as –

```
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 liters.

## 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 statements 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
```

[Live Demo](#)

```
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!
```

## Simple Case Study

Here is a case study if you want to do more practice with class and objects.

Ruby Class Case Study [↗](#)

## Ruby - Variables, Constants and Literals

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 the 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 the global status. It is not recommended to use global variables. They make programs cryptic.

Here is an example showing the usage of global variable.

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

[Live Demo](#)

```
$global_variable = 10
class Class1
  def print_global
    puts "Global variable in Class1 is #{$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 the usage of Instance Variables.

[Live Demo](#)

```
#!/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 the usage of class variable –

[Live Demo](#)

```
#!/usr/bin/ruby

class Customer
  @@no_of_customers = 0
  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
  def total_no_of_customers()
    @@no_of_customers += 1
    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
```

[Live Demo](#)

```

    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 constants. 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 cannot 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 within 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 the 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 floating 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';
```

[Live Demo](#)

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}";
```

[Live Demo](#)

This will produce the following result –

```
Multiplication Value : 86400
```

## Backslash Notations

Following is the list of Backslash notations supported by Ruby –

Notation	Character represented
<code>\n</code>	Newline (0x0a)
<code>\r</code>	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

For more detail on Ruby Strings, go through [Ruby Strings](#) .

## Ruby Arrays

Literals of Ruby Array are created by placing a comma-separated series of object references between the square brackets. A trailing comma is ignored.

### Example


[Live Demo](#)

```
#!/usr/bin/ruby

ary = [ "fred", 10, 3.14, "This is a string", "last element", ]
ary.each do |i|
  puts i
end
```

This will produce the following result –

```
fred
10
3.14
This is a string
last element
```

For more detail on Ruby Arrays, go through [Ruby Arrays](#) .

## Ruby Hashes

A literal Ruby Hash is created by placing a list of key/value pairs between braces, with either a comma or the sequence => between the key and the value. A trailing comma is ignored.

### Example

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

[Live Demo](#)

```
hsh = colors = { "red" => 0xf00, "green" => 0x0f0, "blue" => 0x00f }
hsh.each do |key, value|
  print key, " is ", value, "\n"
end
```

This will produce the following result –

```
red is 3840
green is 240
blue is 15
```

For more detail on Ruby Hashes, go through [Ruby Hashes](#).

## Ruby Ranges

A Range represents an interval which is 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
```

[Live Demo](#)

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

This will produce the following result –

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

For more detail on Ruby Ranges, go through [Ruby Ranges](#).

## Ruby - Operators

Ruby supports a rich set of operators, as you'd expect from a modern language. Most operators are actually method calls. For example, `a + b` is interpreted as `a.+(b)`, where the `+` method in the object referred to by variable `a` is called with `b` as its argument.

For each operator (`+` `-` `*` `/` `%` `**` `&` `|` `^` `<<` `>>` `&&` `||`), there is a corresponding form of abbreviated assignment operator (`+=` `-=` etc.).

## Ruby Arithmetic Operators

Assume variable `a` holds 10 and variable `b` holds 20, then –

Operator	Description	Example
+	Addition – Adds values on either side of the operator.	<code>a + b</code> will give 30



-	Subtraction – Subtracts right hand operand from left hand operand.	a - b will give -10
*	Multiplication – Multiplies values on either side of the operator.	a * b will give 200
/	Division – Divides left hand operand by right hand operand.	b / a will give 2
%	Modulus – Divides left hand operand by right hand operand and returns remainder.	b % a will give 0
**	Exponent – Performs exponential (power) calculation on operators.	a**b will give 10 to the power 20

## Ruby Comparison Operators

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
<=>	Combined comparison operator. Returns 0 if first operand equals second, 1 if first operand is greater than the second and -1 if first operand is less than the second.	(a <=> b) returns -1.
===	Used to test equality within a when clause of a case statement.	(1...10) === 5 returns true.
.eql?	True if the receiver and argument have both the same type and equal values.	1 == 1.0 returns true, but 1.eql?(1.0) is false.
equal?	True if the receiver and argument have the same object id.	if aObj is duplicate of bObj then aObj == bObj is true, a.equal?bObj is false but a.equal?aObj is true.

## Ruby Assignment Operators

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
=	Simple assignment operator, assigns values from right side operands to left side operand.	c = a + b will assign the value of a + b into c
+=	Add AND assignment operator, adds right operand to the left operand and assign the result to left operand.	c += a is equivalent to c = c + a
-=	Subtract AND assignment operator, subtracts right operand from the left operand and assign the result to left operand.	c -= a is equivalent to c = c - a
*=	Multiply AND assignment operator, multiplies right operand with the left operand and assign the result to left operand.	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, divides left operand with the right operand and assign the result to left operand.	c /= a is equivalent to c = c / a
%=	Modulus AND assignment operator, takes modulus using two operands and assign the result to left operand.	c %= a is equivalent to c = c % a
**=	Exponent AND assignment operator, performs exponential (power) calculation on operators and assign value to the left operand.	c **= a is equivalent to c = c ** a

## Ruby Parallel Assignment

Ruby also supports the parallel assignment of variables. This enables multiple variables to be initialized with a single line of Ruby code. For example –

```
a = 10
b = 20
c = 30
```

This may be more quickly declared using parallel assignment –

```
a, b, c = 10, 20, 30
```

Parallel assignment is also useful for swapping the values held in two variables –

```
a, b = b, c
```

## Ruby Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation.

Assume if a = 60; and b = 13; now in binary format they will be as follows –

```
a      = 0011 1100
b      = 0000 1101
-----
a&b    = 0000 1100
a|b    = 0011 1101
a^b    = 0011 0001
~a     = 1100 0011
```

The following Bitwise operators are supported by Ruby language.

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(a & b) will give 12, which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(a   b) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(a ^ b) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~a ) will give -61, which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	a << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 will give 15, which is 0000 1111

## Ruby Logical Operators

The following logical operators are supported by Ruby language

Assume variable *a* holds 10 and variable *b* holds 20, then –

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true, then the condition becomes true.	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero, then the condition becomes true.	(a or b) is true.
&&	Called Logical AND operator. If both the operands are non zero, then the condition becomes true.	(a && b) is true.
	Called Logical OR Operator. If any of the two operands are non zero, then the condition becomes true.	(a    b) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(a && b) is false.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	not(a && b) is false.

## Ruby Ternary Operator

There is one more operator called Ternary Operator. It first evaluates an expression for a true or false value and then executes one of the two given statements depending upon the result of the evaluation. The conditional operator has this syntax –

Operator	Description	Example
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

## Ruby Range Operators

Sequence ranges in Ruby are used to create a range of successive values - consisting of a start value, an end value, and a range of values in between.

In Ruby, these sequences are created using the ".." and "..." range operators. The two-dot form creates an inclusive range, while the three-dot form creates a range that excludes the specified high value.

Operator	Description	Example
..	Creates a range from start point to end point inclusive.	1..10 Creates a range from 1 to 10 inclusive.
...	Creates a range from start point to end point exclusive.	1...10 Creates a range from 1 to 9.

## Ruby defined? Operators

defined? is a special operator that takes the form of a method call to determine whether or not the passed expression is defined. It returns a description string of the expression, or *nil* if the expression isn't defined.

There are various usage of defined? Operator

### Usage 1

```
defined? variable # True if variable is initialized
```

#### For Example

```
foo = 42
defined? foo      # => "local-variable"
defined? $_       # => "global-variable"
defined? bar      # => nil (undefined)
```

### Usage 2

```
defined? method_call # True if a method is defined
```

#### For Example

```
defined? puts      # => "method"
defined? puts(bar)  # => nil (bar is not defined here)
defined? unpack     # => nil (not defined here)
```

### Usage 3

```
# True if a method exists that can be called with super user
defined? super
```

### For Example

```
defined? super    # => "super" (if it can be called)
defined? super    # => nil (if it cannot be)
```

### Usage 4

```
defined? yield    # True if a code block has been passed
```

### For Example

```
defined? yield    # => "yield" (if there is a block passed)
defined? yield    # => nil (if there is no block)
```

## Ruby Dot "." and Double Colon "::" Operators

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

The :: is a unary operator that allows: constants, instance methods and class methods defined within a class or module, to be accessed from anywhere outside the class or module.

**Remember** in Ruby, classes and methods may be considered constants too.

You need to just prefix the :: Const\_name with an expression that returns the appropriate class or module object.

If no prefix expression is used, the main Object class is used by default.

Here are two examples –

```
MR_COUNT = 0          # constant defined on main Object class
module Foo
  MR_COUNT = 0
  ::MR_COUNT = 1      # set global count to 1
  MR_COUNT = 2        # set local count to 2
end
puts MR_COUNT         # this is the global constant
puts Foo::MR_COUNT    # this is the local "Foo" constant
```

### Second Example

```
CONST = ' out there'
class Inside_one
  CONST = proc {' in there'}
  def where_is_my_CONST
    ::CONST + ' inside one'
  end
end
class Inside_two
  CONST = ' inside two'
  def where_is_my_CONST
    CONST
  end
end
puts Inside_one.new.where_is_my_CONST
```

```
puts Inside_two.new.where_is_my_CONST
puts Object::CONST + Inside_two::CONST
puts Inside_two::CONST + CONST
puts Inside_one::CONST
puts Inside_one::CONST.call + Inside_two::CONST
```

## Ruby Operators Precedence

The following table lists all operators from highest precedence to lowest.

Method	Operator	Description
Yes	::	Constant resolution operator
Yes	[] []=	Element reference, element set
Yes	**	Exponentiation (raise to the power)
Yes	! ~ + -	Not, complement, unary plus and minus (method names for the last two are +@ and -@)
Yes	* / %	Multiply, divide, and modulo
Yes	+ -	Addition and subtraction
Yes	>> <<	Right and left bitwise shift
Yes	&	Bitwise 'AND'
Yes	^	Bitwise exclusive 'OR' and regular 'OR'
Yes	<= < > >=	Comparison operators
Yes	<=> == === != =~ !~	Equality and pattern match operators (!= and =~ may not be defined as methods)
	&&	Logical 'AND'
		Logical 'OR'
	.. ...	Range (inclusive and exclusive)
	? :	Ternary if-then-else
	= %= { /= -= +=  = &= >>= <<= *= &&=   = **=	Assignment
	defined?	Check if specified symbol defined
	not	Logical negation
	or and	Logical composition

**NOTE** – Operators with a Yes in the method column are actually methods, and as such may be overridden.

# Ruby - Comments

Comments are lines of annotation within Ruby code that are ignored at runtime. A single line comment starts with # character and they extend from # to the end of the line as follows –

[Live Demo](#)

```
#!/usr/bin/ruby -w
# This is a single line comment.

puts "Hello, Ruby!"
```

When executed, the above program produces the following result –

```
Hello, Ruby!
```

## Ruby Multiline Comments

You can comment multiple lines using **=begin** and **=end** syntax as follows –

[Live Demo](#)

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

puts "Hello, Ruby!"

=begin
This is a multiline comment and can span as many lines as you
like. But =begin and =end should come in the first line only.
=end
```

When executed, the above program produces the following result –

```
Hello, Ruby!
```

Make sure trailing comments are far enough from the code and that they are easily distinguished. If more than one trailing comment exists in a block, align them. For example –

```
@counter      # keeps track times page has been hit
@siteCounter  # keeps track of times all pages have been hit
```

## Ruby - if...else, case, unless

Ruby offers conditional structures that are pretty common to modern languages. Here, we will explain all the conditional statements and modifiers available in Ruby.

### 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

[↗ Live Demo](#)

```
#!/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

[↗ Live Demo](#)

```
#!/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

[↗ Live Demo](#)

```
#!/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

[🔗 Live Demo](#)

```
#!/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

[🔗 Live Demo](#)

```
#!/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 - Loops

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
```

[Live Demo](#)

```
$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
```

[Live Demo](#)

```
$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

[Live Demo](#)

```
#!/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

[Live Demo](#)

```
#!/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

[Live Demo](#)

```
#!/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 the following –

```
(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

[Live Demo](#)

```
#!/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

[Live Demo](#)

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

### Example

[Live Demo](#)

```
#!/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 begin 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 0..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 the 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

[Live Demo](#)

```
#!/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 –

[Live Demo](#)

```
#!/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
```

## Variable Number of Parameters

Suppose you declare a method that takes two parameters, whenever you call this method, you need to pass two parameters along with it.

However, Ruby allows you to declare methods that work with a variable number of parameters. Let us examine a sample of this –

[Live Demo](#)

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

```
def sample (*test)
  puts "The number of parameters is #{test.length}"
  for i in 0...test.length
    puts "The parameters are #{test[i]}"
  end
end
sample "Zara", "6", "F"
sample "Mac", "36", "M", "MCA"
```

In this code, you have declared a method sample that accepts one parameter test. However, this parameter is a variable parameter. This means that this parameter can take in any number of variables. So, the above code will produce the following result –

```
The number of parameters is 3
The parameters are Zara
The parameters are 6
The parameters are F
The number of parameters is 4
```

```
The parameters are Mac
The parameters are 36
The parameters are M
The parameters are MCA
```

## 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 cannot be defined within the method body. The alias of the method keeps 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* cannot 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 - Blocks

You have seen how Ruby defines methods where you can put number of statements and then you call that method. Similarly, Ruby has a concept of Block.

- A block consists of chunks of code.
- You assign a name to a block.
- The code in the block is always enclosed within braces ({}).
- A block is always invoked from a function with the same name as that of the block. This means that if you have a block with the name *test*, then you use the function *test* to invoke this block.
- You invoke a block by using the *yield* statement.

## Syntax

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

Here, you will learn to invoke a block by using a simple *yield* statement. You will also learn to use a *yield* statement with parameters for invoking a block. You will check the sample code with both types of *yield* statements.

## The yield Statement

Let's look at an example of the yield statement –

```
#!/usr/bin/ruby  
  
def test  
  puts "You are in the method"  
  yield  
  puts "You are again back to the method"  
  yield  
end  
test {puts "You are in the block"}
```

[Live Demo](#)

This will produce the following result –

```
You are in the method
You are in the block
You are again back to the method
You are in the block
```

You also can pass parameters with the `yield` statement. Here is an example –

```
#!/usr/bin/ruby

def test
  yield 5
  puts "You are in the method test"
  yield 100
end
test {|i| puts "You are in the block #{i}"}
```

[Live Demo](#)

This will produce the following result –

```
You are in the block 5
You are in the method test
You are in the block 100
```

Here, the `yield` statement is written followed by parameters. You can even pass more than one parameter. In the block, you place a variable between two vertical lines (`|`) to accept the parameters. Therefore, in the preceding code, the `yield 5` statement passes the value 5 as a parameter to the test block.

Now, look at the following statement –

```
test {|i| puts "You are in the block #{i}"}
```

Here, the value 5 is received in the variable `i`. Now, observe the following `puts` statement –

```
puts "You are in the block #{i}"
```

The output of this `puts` statement is –

```
You are in the block 5
```

If you want to pass more than one parameters, then the `yield` statement becomes –

```
yield a, b
```

and the block is –

```
test {|a, b| statement}
```

The parameters will be separated by commas.

## Blocks and Methods

You have seen how a block and a method can be associated with each other. You normally invoke a block by using the `yield` statement from a method that has the same name as that of the block. Therefore, you write –

[Live Demo](#)

```
#!/usr/bin/ruby

def test
  yield
end
test{ puts "Hello world"}
```

This example is the simplest way to implement a block. You call the test block by using the *yield* statement.

But if the last argument of a method is preceded by *&*, then you can pass a block to this method and this block will be assigned to the last parameter. In case both *\** and *&* are present in the argument list, *&* should come later.

```
#!/usr/bin/ruby

def test(&block)
  block.call
end
test { puts "Hello World!"}
```

[Live Demo](#)

This will produce the following result –

```
Hello World!
```

## BEGIN and END Blocks

Every Ruby source file can declare blocks of code to be run as the file is being loaded (the BEGIN blocks) and after the program has finished executing (the END blocks).

```
#!/usr/bin/ruby

BEGIN {
  # BEGIN block code
  puts "BEGIN code block"
}

END {
  # END block code
  puts "END code block"
}

# MAIN block code
puts "MAIN code block"
```

[Live Demo](#)

A program may include multiple BEGIN and END blocks. BEGIN blocks are executed in the order they are encountered. END blocks are executed in reverse order. When executed, the above program produces the following result –

```
BEGIN code block
MAIN code block
END code block
```

## Ruby - Modules and Mixins

Modules are a way of grouping together methods, classes, and constants. Modules give you two major benefits.

- ▣ Modules provide a *namespace* and prevent name clashes.
- ▣ Modules implement the *mixin* facility.

Modules define a namespace, a sandbox in which your methods and constants can play without having to worry about being stepped on by other methods and constants.

## Syntax

```
module Identifier
  statement1
  statement2
  .....
end
```

Module constants are named just like class constants, with an initial uppercase letter. The method definitions look similar, too: Module methods are defined just like class methods.

As with class methods, you call a module method by preceding its name with the module's name and a period, and you reference a constant using the module name and two colons.

## Example

```
#!/usr/bin/ruby

# Module defined in trig.rb file

module Trig
  PI = 3.141592654
  def Trig.sin(x)
    # ..
  end
  def Trig.cos(x)
    # ..
  end
end
```

We can define one more module with the same function name but different functionality –

```
#!/usr/bin/ruby

# Module defined in moral.rb file

module Moral
  VERY_BAD = 0
  BAD = 1
  def Moral.sin(badness)
    # ...
  end
end
```

Like class methods, whenever you define a method in a module, you specify the module name followed by a dot and then the method name.

## Ruby require Statement

The require statement is similar to the include statement of C and C++ and the import statement of Java. If a third program wants to use any defined module, it can simply load the module files using the Ruby *require* statement –

## Syntax

```
require filename
```

Here, it is not required to give **.rb** extension along with a file name.

## Example

```
$LOAD_PATH << '.'

require 'trig.rb'
require 'moral'

y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::VERY_BAD)
```

Here we are using **\$LOAD\_PATH << '.'** to make Ruby aware that included files must be searched in the current directory. If you do not want to use \$LOAD\_PATH then you can use **require\_relative** to include files from a relative directory.

**IMPORTANT** – Here, both the files contain the same function name. So, this will result in code ambiguity while including in calling program but modules avoid this code ambiguity and we are able to call appropriate function using module name.

## Ruby include Statement

You can embed a module in a class. To embed a module in a class, you use the *include* statement in the class –

### Syntax

```
include modulename
```

If a module is defined in a separate file, then it is required to include that file using *require* statement before embedding module in a class.

## Example

Consider the following module written in *support.rb* file.

```
module Week
  FIRST_DAY = "Sunday"
  def Week.weeks_in_month
    puts "You have four weeks in a month"
  end
  def Week.weeks_in_year
    puts "You have 52 weeks in a year"
  end
end
```

Now, you can include this module in a class as follows –

```
#!/usr/bin/ruby
$LOAD_PATH << '.'
require "support"

class Decade
  include Week
  no_of_yrs = 10
  def no_of_months
    puts Week::FIRST_DAY
    number = 10*12
  end
end
```



```

        puts number
    end
end
d1 = Decade.new
puts Week::FIRST_DAY
Week.weeks_in_month
Week.weeks_in_year
d1.no_of_months

```

This will produce the following result –

```

Sunday
You have four weeks in a month
You have 52 weeks in a year
Sunday
120

```

## Mixins in Ruby

Before going through this section, we assume you have the knowledge of Object Oriented Concepts.

When a class can inherit features from more than one parent class, the class is supposed to show multiple inheritance.

Ruby does not support multiple inheritance directly but Ruby Modules have another wonderful use. At a stroke, they pretty much eliminate the need for multiple inheritance, providing a facility called a *mixin*.

Mixins give you a wonderfully controlled way of adding functionality to classes. However, their true power comes out when the code in the mixin starts to interact with code in the class that uses it.

Let us examine the following sample code to gain an understand of mixin –

```

module A
  def a1
  end
  def a2
  end
end
module B
  def b1
  end
  def b2
  end
end

class Sample
  include A
  include B
  def s1
  end
end

samp = Sample.new
samp.a1
samp.a2
samp.b1
samp.b2
samp.s1

```

Module A consists of the methods a1 and a2. Module B consists of the methods b1 and b2. The class Sample includes both modules A and B. The class Sample can access all four methods, namely, a1, a2, b1, and b2. Therefore, you can see that the class Sample inherits from both the modules. Thus, you can say the class Sample shows multiple inheritance or a *mixin*.

## Ruby - Strings

A String object in Ruby holds and manipulates an arbitrary sequence of one or more bytes, typically representing characters that represent human language.

The simplest string literals are enclosed in single quotes (the apostrophe character). The text within the quote marks is the value of the string –

```
'This is a simple Ruby string literal'
```

If you need to place an apostrophe within a single-quoted string literal, precede it with a backslash, so that the Ruby interpreter does not think that it terminates the string –

```
'Won\'t you read O\'Reilly\'s book?'
```

The backslash also works to escape another backslash, so that the second backslash is not itself interpreted as an escape character.

Following are the string-related features of Ruby.

### Expression Substitution

Expression substitution is a means of embedding the value of any Ruby expression into a string using #{ and } –

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

[Live Demo](#)

```
x, y, z = 12, 36, 72
puts "The value of x is #{ x }."
puts "The sum of x and y is #{ x + y }."
puts "The average was #{ (x + y + z)/3 }."
```

This will produce the following result –

```
The value of x is 12.
The sum of x and y is 48.
The average was 40.
```

### General Delimited Strings

With general delimited strings, you can create strings inside a pair of matching though arbitrary delimiter characters, e.g., !, (, {, <, etc., preceded by a percent character (%). Q, q, and x have special meanings. General delimited strings can be –

```
%{Ruby is fun.} equivalent to "Ruby is fun."
%Q{ Ruby is fun. } equivalent to " Ruby is fun. "
%q[Ruby is fun.] equivalent to a single-quoted string
%x!ls! equivalent to back tick command output `ls`
```

## Escape Characters

Following table is a list of escape or non-printable characters that can be represented with the backslash notation.

## Character Encoding

The default character set for Ruby is ASCII, whose characters may be represented by single bytes. If you use UTF-8, or another modern character set, characters may be represented in one to four bytes.

You can change your character set using \$KCODE at the beginning of your program, like this –

```
$KCODE = 'u'
```

Following are the possible values for \$KCODE.

## String Built-in Methods

We need to have an instance of String object to call a String method. Following is the way to create an instance of String object –

```
new [String.new(str = "")]
```

This will return a new string object containing a copy of *str*. Now, using *str* object, we can all use any available instance methods. For example –

```
#!/usr/bin/ruby

myStr = String.new("THIS IS TEST")
foo = myStr.downcase

puts "#{foo}"
```

[Live Demo](#)

This will produce the following result –

```
this is test
```

Following are the public String methods ( Assuming str is a String object ) –

## String unpack Directives

Following table lists the unpack directives for method String#unpack.

### Example

Try the following example to unpack various data.

```
"abc \0\0abc \0\0".unpack('A6Z6')    #=> ["abc", "abc "]
"abc \0\0".unpack('a3a3')             #=> ["abc", " \000\000"]
"abc \0abc \0".unpack('Z*Z*')         #=> ["abc ", "abc "]
"aa".unpack('b8B8')                   #=> ["10000110", "01100001"]
"aaa".unpack('h2H2c')                 #=> ["16", "61", 97]
```

```
"\xfe\xff\xfe\xff".unpack('sS')    #=> [-2, 65534]
"now = 20is".unpack('M*')           #=> ["now is"]
"whole".unpack('xax2aX2aX1aX2a')    #=> ["h", "e", "l", "l", "o"]
```

## 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
```

[Live Demo](#)

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}"
```

[Live Demo](#)

This will produce the following result –

```
["mac", "mac", "mac", "mac"]
```

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}"
```

[Live Demo](#)

This will produce the following result –

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

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}"
```

[Live Demo](#)

This will produce the following result –

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## Array Built-in Methods

We need to have an instance of Array object to call an Array method. As we have seen, following is the way to create an instance of Array object –

```
Array.[](...) [or] Array[...] [or] [...]
```

This will return a new array populated with the given objects. Now, using the created object, we can call any available instance methods. For example –

```
#!/usr/bin/ruby  
  
digits = Array(0..9)  
num = digits.at(6)  
puts "#{num}"
```

[Live Demo](#)

This will produce the following result –

```
6
```

Following are the public array methods (assuming array is an array object) –

## Array pack Directives

Following table lists the pack directives for use with Array#pack.

### Example

Try the following example to pack various data.

```
a = [ "a", "b", "c" ]
n = [ 65, 66, 67 ]
puts a.pack("A3A3A3")    #=> "a b c "
puts a.pack("a3a3a3")    #=> "a\000\000b\000\000c\000\000"
puts n.pack("ccc")       #=> "ABC"
```

[Live Demo](#)

This will produce the following result –

```
a b c
abc
ABC
```

## 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]}"
```

[Live Demo](#)

This will produce the following result –

```
month
month
```

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

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

```
puts "#{H['a']}"
puts "#{H['b']}"
```

[Live Demo](#)

This will produce the following result –

```
100
200
```

You can use any Ruby object as a key or value, even an array, so the following example is a valid one –

```
[1, "jan"] => "January"
```

## Hash Built-in Methods

We need to have an instance of Hash object to call a Hash method. As we have seen, following is the way to create an instance of Hash object –

```
Hash[[key =>|, value]* ] or
Hash.new [or] Hash.new(obj) [or]
Hash.new { |hash, key| block }
```

This will return a new hash populated with the given objects. Now using the created object, we can call any available instance methods. For example –

```
#!/usr/bin/ruby

$, = ", "
months = Hash.new( "month" )
months = {"1" => "January", "2" => "February"}

keys = months.keys
puts "#{keys}"
```

[Live Demo](#)

This will produce the following result –

```
["1", "2"]
```

Following are the public hash methods (assuming *hash* is an array object) –

Sr.No.	Methods & Description
1	<b>hash == other_hash</b> Tests whether two hashes are equal, based on whether they have the same number of key-value pairs, and whether the key-value pairs match the corresponding pair in each hash.
2	<b>hash[key]</b> Using a key, references a value from hash. If the key is not found, returns a default value.
3	<b>hash[key] = value</b> Associates the value given by <i>value</i> with the key given by <i>key</i> .
4	<b>hash.clear</b>

	Removes all key-value pairs from hash.
5	<b>hash.default(key = nil)</b> Returns the default value for <i>hash</i> , nil if not set by default=. ([] returns a default value if the key does not exist in <i>hash</i> .)
6	<b>hash.default = obj</b> Sets a default value for <i>hash</i> .
7	<b>hash.default_proc</b> Returns a block if <i>hash</i> was created by a block.
8	<b>hash.delete(key) [or]</b> <b>array.delete(key) {  key  block }</b> Deletes a key-value pair from <i>hash</i> by <i>key</i> . If block is used, returns the result of a block if pair is not found. Compare <i>delete_if</i> .
9	<b>hash.delete_if {  key,value  block }</b> Deletes a key-value pair from <i>hash</i> for every pair the block evaluates to <i>true</i> .
10	<b>hash.each {  key,value  block }</b> Iterates over <i>hash</i> , calling the block once for each key, passing the key-value as a two-element array.
11	<b>hash.each_key {  key  block }</b> Iterates over <i>hash</i> , calling the block once for each key, passing <i>key</i> as a parameter.
12	<b>hash.each_key {  key_value_array  block }</b> Iterates over <i>hash</i> , calling the block once for each <i>key</i> , passing the <i>key</i> and <i>value</i> as parameters.
13	<b>hash.each_key {  value  block }</b> Iterates over <i>hash</i> , calling the block once for each <i>key</i> , passing <i>value</i> as a parameter.
14	<b>hash.empty?</b> Tests whether hash is empty (contains no key-value pairs), returning <i>true</i> or <i>false</i> .
15	<b>hash.fetch(key [, default] ) [or]</b> <b>hash.fetch(key) {   key   block }</b>



	Returns a value from <i>hash</i> for the given <i>key</i> . If the <i>key</i> can't be found, and there are no other arguments, it raises an <i>IndexError</i> exception; if <i>default</i> is given, it is returned; if the optional block is specified, its result is returned.
16	<b>hash.has_key?(key) [or] hash.include?(key) [or]</b> <b>hash.key?(key) [or] hash.member?(key)</b> Tests whether a given <i>key</i> is present in hash, returning <i>true</i> or <i>false</i> .
17	<b>hash.has_value?(value)</b> Tests whether hash contains the given <i>value</i> .
18	<b>hash.index(value)</b> Returns the <i>key</i> for the given <i>value</i> in hash, <i>nil</i> if no matching value is found.
19	<b>hash.indexes(keys)</b> Returns a new array consisting of values for the given key(s). Will insert the default value for keys that are not found. This method is deprecated. Use <i>select</i> .
20	<b>hash.indices(keys)</b> Returns a new array consisting of values for the given key(s). Will insert the default value for keys that are not found. This method is deprecated. Use <i>select</i> .
21	<b>hash.inspect</b> Returns a pretty print string version of hash.
22	<b>hash.invert</b> Creates a new <i>hash</i> , inverting <i>keys</i> and <i>values</i> from <i>hash</i> ; that is, in the new hash, the keys from <i>hash</i> become values and values become keys.
23	<b>hash.keys</b> Creates a new array with keys from <i>hash</i> .
24	<b>hash.length</b> Returns the size or length of <i>hash</i> as an integer.
25	<b>hash.merge(other_hash) [or]</b> <b>hash.merge(other_hash) {  key, oldval, newval  block }</b>

	Returns a new hash containing the contents of <i>hash</i> and <i>other_hash</i> , overwriting pairs in <i>hash</i> with duplicate keys with those from <i>other_hash</i> .
26	<b>hash.merge!(other_hash) [or]</b> <b>hash.merge!(other_hash) {  key, oldval, newval  block }</b> Same as <i>merge</i> , but changes are done in place.
27	<b>hash.rehash</b> Rebuilds <i>hash</i> based on the current values for each <i>key</i> . If values have changed since they were inserted, this method reindexes <i>hash</i> .
28	<b>hash.reject {  key, value  block }</b> Creates a new <i>hash</i> for every pair the <i>block</i> evaluates to <i>true</i>
29	<b>hash.reject! {  key, value  block }</b> Same as <i>reject</i> , but changes are made in place.
30	<b>hash.replace(other_hash)</b> Replaces the contents of <i>hash</i> with the contents of <i>other_hash</i> .
31	<b>hash.select {  key, value  block }</b> Returns a new array consisting of key-value pairs from <i>hash</i> for which the <i>block</i> returns <i>true</i> .
32	<b>hash.shift</b> Removes a key-value pair from <i>hash</i> , returning it as a two-element array.
33	<b>hash.size</b> Returns the <i>size</i> or length of <i>hash</i> as an integer.
34	<b>hash.sort</b> Converts <i>hash</i> to a two-dimensional array containing arrays of key-value pairs, then sorts it as an array.
35	<b>hash.store(key, value)</b> Stores a key-value pair in <i>hash</i> .
36	<b>hash.to_a</b>

	Creates a two-dimensional array from hash. Each key/value pair is converted to an array, and all these arrays are stored in a containing array.
37	<b>hash.to_hash</b> Returns <i>hash</i> (self).
38	<b>hash.to_s</b> Converts <i>hash</i> to an array, then converts that array to a string.
39	<b>hash.update(other_hash) [or]</b> <b>hash.update(other_hash) { key, oldval, newval  block}</b> Returns a new hash containing the contents of <i>hash</i> and <i>other_hash</i> , overwriting pairs in <i>hash</i> with duplicate keys with those from <i>other_hash</i> .
40	<b>hash.value?(value)</b> Tests whether <i>hash</i> contains the given <i>value</i> .
41	<b>hash.values</b> Returns a new array containing all the values of <i>hash</i> .
42	<b>hash.values_at(obj, ...)</b> Returns a new array containing the values from <i>hash</i> that are associated with the given key or keys.

## Ruby - Date & Time

The **Time** class represents dates and times in Ruby. It is a thin layer over the system date and time functionality provided by the operating system. This class may be unable on your system to represent dates before 1970 or after 2038.

This chapter makes you familiar with all the most wanted concepts of date and time.

### Getting Current Date and Time

Following is the simple example to get current date and time –

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

time1 = Time.new
puts "Current Time : " + time1.inspect

# Time.now is a synonym:
time2 = Time.now
puts "Current Time : " + time2.inspect
```

[↗ Live Demo](#)

This will produce the following result –

```
Current Time : Mon Jun 02 12:02:39 -0700 2008
Current Time : Mon Jun 02 12:02:39 -0700 2008
```

## Getting Components of a Date & Time

We can use *Time* object to get various components of date and time. Following is the example showing the same –

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

time = Time.new

# Components of a Time
puts "Current Time : " + time.inspect
puts time.year      # => Year of the date
puts time.month     # => Month of the date (1 to 12)
puts time.day       # => Day of the date (1 to 31 )
puts time.wday      # => 0: Day of week: 0 is Sunday
puts time.yday      # => 365: Day of year
puts time.hour      # => 23: 24-hour clock
puts time.min       # => 59
puts time.sec       # => 59
puts time.usec      # => 999999: microseconds
puts time.zone      # => "UTC": timezone name
```

[Live Demo](#)

This will produce the following result –

```
Current Time : Mon Jun 02 12:03:08 -0700 2008
2008
6
2
1
154
12
3
8
247476
UTC
```

## Time.utc, Time.gm and Time.local Functions

These two functions can be used to format date in a standard format as follows –

```
# July 8, 2008
Time.local(2008, 7, 8)
# July 8, 2008, 09:10am, local time
Time.local(2008, 7, 8, 9, 10)
# July 8, 2008, 09:10 UTC
Time.utc(2008, 7, 8, 9, 10)
# July 8, 2008, 09:10:11 GMT (same as UTC)
Time.gm(2008, 7, 8, 9, 10, 11)
```

Following is the example to get all the components in an array in the following format –

```
[sec,min,hour,day,month,year,wday,yday,isdst,zone]
```

Try the following –

[Live Demo](#)

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

time = Time.new
values = time.to_a
p values
```

This will generate the following result –

```
[26, 10, 12, 2, 6, 2008, 1, 154, false, "MST"]
```

This array could be passed to *Time.utc* or *Time.local* functions to get different format of dates as follows –

[Live Demo](#)

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

time = Time.new
values = time.to_a
puts Time.utc(*values)
```

This will generate the following result –

```
Mon Jun 02 12:15:36 UTC 2008
```

Following is the way to get time represented internally as seconds since the (platform-dependent) epoch –

```
# Returns number of seconds since epoch
time = Time.now.to_i

# Convert number of seconds into Time object.
Time.at(time)

# Returns second since epoch which includes microseconds
time = Time.now.to_f
```

## Timezones and Daylight Savings Time

You can use a *Time* object to get all the information related to Timezones and daylight savings as follows –

```
time = Time.new

# Here is the interpretation
time.zone          # => "UTC": return the timezone
time.utc_offset    # => 0: UTC is 0 seconds offset from UTC
time.zone          # => "PST" (or whatever your timezone is)
time.isdst         # => false: If UTC does not have DST.
time.utc?          # => true: if t is in UTC time zone
time.localtime     # Convert to local timezone.
time.gmtime        # Convert back to UTC.
time.getlocal      # Return a new Time object in local zone
time.getutc        # Return a new Time object in UTC
```

## Formatting Times and Dates

There are various ways to format date and time. Here is one example showing a few –

[Live Demo](#)

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

```
time = Time.new
puts time.to_s
puts time.ctime
puts time.localtime
puts time.strftime("%Y-%m-%d %H:%M:%S")
```

This will produce the following result –

```
Mon Jun 02 12:35:19 -0700 2008
Mon Jun  2 12:35:19 2008
Mon Jun 02 12:35:19 -0700 2008
2008-06-02 12:35:19
```

## Time Formatting Directives

These directives in the following table are used with the method *Time.strftime*.

Sr.No.	Directive & Description
1	<b>%a</b> The abbreviated weekday name (Sun).
2	<b>%A</b> The full weekday name (Sunday).
3	<b>%b</b> The abbreviated month name (Jan).
4	<b>%B</b> The full month name (January).
5	<b>%c</b> The preferred local date and time representation.
6	<b>%d</b> Day of the month (01 to 31).
7	<b>%H</b> Hour of the day, 24-hour clock (00 to 23).
8	<b>%I</b> Hour of the day, 12-hour clock (01 to 12).

9	<b>%j</b> Day of the year (001 to 366).
10	<b>%m</b> Month of the year (01 to 12).
11	<b>%M</b> Minute of the hour (00 to 59).
12	<b>%p</b> Meridian indicator (AM or PM).
13	<b>%S</b> Second of the minute (00 to 60).
14	<b>%U</b> Week number of the current year, starting with the first Sunday as the first day of the first week (00 to 53).
15	<b>%W</b> Week number of the current year, starting with the first Monday as the first day of the first week (00 to 53).
16	<b>%w</b> Day of the week (Sunday is 0, 0 to 6).
17	<b>%x</b> Preferred representation for the date alone, no time.
18	<b>%X</b> Preferred representation for the time alone, no date.
19	<b>%y</b> Year without a century (00 to 99).
20	<b>%Y</b> Year with century.

21	<b>%Z</b> Time zone name.
22	<b>%%</b> Literal % character.

## Time Arithmetic

You can perform simple arithmetic with time as follows –

[Live Demo](#)

```
now = Time.now           # Current time
puts now

past = now - 10          # 10 seconds ago. Time - number => Time
puts past

future = now + 10        # 10 seconds from now Time + number => Time
puts future

diff = future - now      # => 10 Time - Time => number of seconds
puts diff
```

This will produce the following result –

```
Thu Aug 01 20:57:05 -0700 2013
Thu Aug 01 20:56:55 -0700 2013
Thu Aug 01 20:57:15 -0700 2013
10.0
```

## Ruby - Ranges

Ranges occur everywhere: January to December, 0 to 9, lines 50 through 67, and so on. Ruby supports ranges and allows us to use ranges in a variety of ways –

- ▣ Ranges as Sequences
- ▣ Ranges as Conditions
- ▣ Ranges as Intervals

### Ranges as Sequences

The first and perhaps the most natural use of ranges is to express a sequence. Sequences have a start point, an end point, and a way to produce successive values in the sequence.

Ruby creates these sequences using the `".."` and `"..."` range operators. The two-dot form creates an inclusive range, while the three-dot form creates a range that excludes the specified high value.

```
(1..5)      #==> 1, 2, 3, 4, 5
(1...5)     #==> 1, 2, 3, 4
('a'..'d')  #==> 'a', 'b', 'c', 'd'
```



The sequence 1..100 is held as a Range *object* containing references to two *Fixnum* objects. If you need to, you can convert a range to a list using the `to_a` method. Try the following example –

[Live Demo](#)

```
#!/usr/bin/ruby

$, =", " # Array value separator
range1 = (1..10).to_a
range2 = ('bar'..'bat').to_a

puts "#{range1}"
puts "#{range2}"
```

This will produce the following result –

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
["bar", "bas", "bat"]
```

Ranges implement methods that let you iterate over them and test their contents in a variety of ways –

[Live Demo](#)

```
#!/usr/bin/ruby

# Assume a range
digits = 0..9

puts digits.include?(5)
ret = digits.min
puts "Min value is #{ret}"

ret = digits.max
puts "Max value is #{ret}"

ret = digits.reject {|i| i < 5 }
puts "Rejected values are #{ret}"

digits.each do |digit|
  puts "In Loop #{digit}"
end
```

This will produce the following result –

```
true
Min value is 0
Max value is 9
Rejected values are 5, 6, 7, 8, 9
In Loop 0
In Loop 1
In Loop 2
In Loop 3
In Loop 4
In Loop 5
In Loop 6
In Loop 7
In Loop 8
In Loop 9
```

## Ranges as Conditions

Ranges may also be used as conditional expressions. For example, the following code fragment prints sets of lines from the standard input, where the first line in each set contains the word *start* and the last line the word *ends* –

```
while gets
  print if /start/../end/
end
```

Ranges can be used in case statements –

```
#!/usr/bin/ruby

score = 70

result = case score
  when 0..40 then "Fail"
  when 41..60 then "Pass"
  when 61..70 then "Pass with Merit"
  when 71..100 then "Pass with Distinction"
  else "Invalid Score"
end

puts result
```

[Live Demo](#)

This will produce the following result –

```
Pass with Merit
```

## Ranges as Intervals

A final use of the versatile range is as an interval test: seeing if some value falls within the interval represented by the range. This is done using `===`, the case equality operator.

```
#!/usr/bin/ruby

if ((1..10) === 5)
  puts "5 lies in (1..10)"
end

if (('a'..'j') === 'c')
  puts "c lies in ('a'..'j')"
end

if (('a'..'j') === 'z')
  puts "z lies in ('a'..'j')"
end
```

[Live Demo](#)

This will produce the following result –

```
5 lies in (1..10)
c lies in ('a'..'j')
```

## Ruby - Iterators

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
```

[↗ Live Demo](#)

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  
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
```

[Live Demo](#)

This will produce the following result –

```
10
20
30
40
50
```

## Ruby - File I/O

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 the previous chapters, you have 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
```

[Live Demo](#)

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 puts 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
```

[Live Demo](#)

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"
```

[Live Demo](#)

This will produce the following result –

## 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 –

## 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| puts ch; }
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 –

## 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 the 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( "test.txt" ) # => Fri May 09 10:44:44 -0700 2008
File::atime( "test.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 *rw-r-x-r-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* performs 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
```

## Built-in Functions

Here are the ruby built-in functions to process files and directories –

- ▣ [File Class and Methods](#) .
- ▣ [Dir Class and Methods](#) .

## Ruby - Exceptions

The execution and the exception always go together. If you are opening a file, which does not exist, then if you did not handle this situation properly, then your program is considered to be of bad quality.

The program stops if an exception occurs. So exceptions are used to handle various type of errors, which may occur during a program execution and take appropriate action instead of halting program completely.

Ruby provide a nice mechanism to handle exceptions. We enclose the code that could raise an exception in a *begin/end* block and use *rescue* clauses to tell Ruby the types of exceptions we want to handle.

## Syntax

```
begin
# -
rescue OneTypeOfException
# -
rescue AnotherTypeOfException
# -
else
# Other exceptions
ensure
```

```
# Always will be executed
end
```

Everything from *begin* to *rescue* is protected. If an exception occurs during the execution of this block of code, control is passed to the block between *rescue* and *end*.

For each *rescue* clause in the *begin* block, Ruby compares the raised Exception against each of the parameters in turn. The match will succeed if the exception named in the rescue clause is the same as the type of the currently thrown exception, or is a superclass of that exception.

In an event that an exception does not match any of the error types specified, we are allowed to use an *else* clause after all the *rescue* clauses.

## Example

[🔗 Live Demo](#)

```
#!/usr/bin/ruby

begin
  file = open("/unexistant_file")
  if file
    puts "File opened successfully"
  end
rescue
  file = STDIN
end
print file, "==", STDIN, "\n"
```

This will produce the following result. You can see that *STDIN* is substituted to *file* because *open* failed.

```
#<IO:0xb7d16f84>==#<IO:0xb7d16f84>
```

## Using retry Statement

You can capture an exception using *rescue* block and then use *retry* statement to execute *begin* block from the beginning.

### Syntax

```
begin
  # Exceptions raised by this code will
  # be caught by the following rescue clause
rescue
  # This block will capture all types of exceptions
  retry # This will move control to the beginning of begin
end
```

## Example

```
#!/usr/bin/ruby

begin
  file = open("/unexistant_file")
  if file
    puts "File opened successfully"
  end
rescue
```

```
fname = "existant_file"
retry
end
```

The following is the flow of the process –

- ▣ An exception occurred at open.
- ▣ Went to rescue. fname was re-assigned.
- ▣ By retry went to the beginning of the begin.
- ▣ This time file opens successfully.
- ▣ Continued the essential process.

**NOTE** – Notice that if the file of re-substituted name does not exist this example code retries infinitely. Be careful if you use *retry* for an exception process.

## Using raise Statement

You can use *raise* statement to raise an exception. The following method raises an exception whenever it's called. It's second message will be printed.

### Syntax

```
raise

OR

raise "Error Message"

OR

raise ExceptionType, "Error Message"

OR

raise ExceptionType, "Error Message" condition
```

The first form simply re-raises the current exception (or a `RuntimeError` if there is no current exception). This is used in exception handlers that need to intercept an exception before passing it on.

The second form creates a new *RuntimeError* exception, setting its message to the given string. This exception is then raised up the call stack.

The third form uses the first argument to create an exception and then sets the associated message to the second argument.

The fourth form is similar to the third form but you can add any conditional statement like *unless* to raise an exception.

### Example

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

[Live Demo](#)

```
begin
  puts 'I am before the raise.'
  raise 'An error has occurred.'
  puts 'I am after the raise.'
rescue
  puts 'I am rescued.'
end
puts 'I am after the begin block.'
```

This will produce the following result –

```
I am before the raise.
I am rescued.
I am after the begin block.
```

One more example showing the usage of *raise* –

```
#!/usr/bin/ruby

begin
  raise 'A test exception.'
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
end
```

[Live Demo](#)

This will produce the following result –

```
A test exception.
["main.rb:4"]
```

## Using ensure Statement

Sometimes, you need to guarantee that some processing is done at the end of a block of code, regardless of whether an exception was raised. For example, you may have a file open on entry to the block and you need to make sure it gets closed as the block exits.

The *ensure* clause does just this. *ensure* goes after the last *rescue* clause and contains a chunk of code that will always be executed as the block terminates. It doesn't matter if the block exits normally, if it raises and rescues an exception, or if it is terminated by an uncaught exception, the *ensure* block will get run.

## Syntax

```
begin
  #.. process
  #..raise exception
rescue
  #.. handle error
ensure
  #.. finally ensure execution
  #.. This will always execute.
end
```

## Example

[Live Demo](#)

```
begin
  raise 'A test exception.'
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
ensure
  puts "Ensuring execution"
end
```

This will produce the following result –

```
A test exception.
["main.rb:4"]
Ensuring execution
```

## Using else Statement

If the *else* clause is present, it goes after the *rescue* clauses and before any *ensure*.

The body of an *else* clause is executed only if no exceptions are raised by the main body of code.

## Syntax

```
begin
  #.. process
  #..raise exception
rescue
  # .. handle error
else
  #.. executes if there is no exception
ensure
  #.. finally ensure execution
  #.. This will always execute.
end
```

## Example

[Live Demo](#)

```
begin
  # raise 'A test exception.'
  puts "I'm not raising exception"
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
else
  puts "Congratulations-- no errors!"
ensure
  puts "Ensuring execution"
end
```

This will produce the following result –

```
I'm not raising exception
Congratulations-- no errors!
Ensuring execution
```

Raised error message can be captured using `$!` variable.

# Catch and Throw

While the exception mechanism of `raise` and `rescue` is great for abandoning the execution when things go wrong, it's sometimes nice to be able to jump out of some deeply nested construct during normal processing. This is where `catch` and `throw` come in handy.

The *catch* defines a block that is labeled with the given name (which may be a `Symbol` or a `String`). The block is executed normally until a `throw` is encountered.

## Syntax

```
throw :lablename
#.. this will not be executed
catch :lablename do
#.. matching catch will be executed after a throw is encountered.
end

OR

throw :lablename condition
#.. this will not be executed
catch :lablename do
#.. matching catch will be executed after a throw is encountered.
end
```

## Example

The following example uses a `throw` to terminate interaction with the user if `!` is typed in response to any prompt.

```
def promptAndGet(prompt)
  print prompt
  res = readline.chomp
  throw :quitRequested if res == "!"
  return res
end

catch :quitRequested do
  name = promptAndGet("Name: ")
  age = promptAndGet("Age: ")
  sex = promptAndGet("Sex: ")
  # ..
  # process information
end
promptAndGet("Name: ")
```

You should try the above program on your machine because it needs manual interaction. This will produce the following result –

```
Name: Ruby on Rails
Age: 3
Sex: !
Name:Just Ruby
```

## Class Exception



Ruby's standard classes and modules raise exceptions. All the exception classes form a hierarchy, with the class `Exception` at the top. The next level contains seven different types –

- `Interrupt`
- `NoMemoryError`
- `SignalException`
- `ScriptError`
- `StandardError`
- `SystemExit`

There is one other exception at this level, **Fatal**, but the Ruby interpreter only uses this internally.

Both `ScriptError` and `StandardError` have a number of subclasses, but we do not need to go into the details here. The important thing is that if we create our own exception classes, they need to be subclasses of either class `Exception` or one of its descendants.

Let's look at an example –

```
class FileSaveError < StandardError
  attr_reader :reason
  def initialize(reason)
    @reason = reason
  end
end
```

Now, look at the following example, which will use this exception –

```
File.open(path, "w") do |file|
  begin
    # Write out the data ...
  rescue
    # Something went wrong!
    raise FileSaveError.new($!)
  end
end
```

The important line here is `raise FileSaveError.new($!)`. We call `raise` to signal that an exception has occurred, passing it a new instance of `FileSaveError`, with the reason being that specific exception caused the writing of the data to fail.

## Ruby - Object Oriented

Ruby is a pure object-oriented language and everything appears to Ruby as an object. Every value in Ruby is an object, even the most primitive things: strings, numbers and even `true` and `false`. Even a class itself is an *object* that is an instance of the `Class` class. This chapter will take you through all the major functionalities related to Object Oriented Ruby.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and methods within a class are called members of the class.

### Ruby Class Definition

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the **class name** and is delimited with an **end**. For example, we defined the Box class using the keyword class as follows –

```
class Box
  code
end
```

The name must begin with a capital letter and by convention names that contain more than one word are run together with each word capitalized and no separating characters (CamelCase).

## Define Ruby Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class using **new** keyword. Following statements declare two objects of class Box –

```
box1 = Box.new
box2 = Box.new
```

## The initialize Method

The **initialize method** is a standard Ruby class method and works almost same way as **constructor** works in other object oriented programming languages. The initialize method is useful when you want to initialize some class variables at the time of object creation. This method may take a list of parameters and like any other ruby method it would be preceded by **def** keyword as shown below –

```
class Box
  def initialize(w,h)
    @width, @height = w, h
  end
end
```

## The instance Variables

The **instance variables** are kind of class attributes and they become properties of objects once objects are created using the class. Every object's attributes are assigned individually and share no value with other objects. They are accessed using the **@** operator within the class but to access them outside of the class we use **public** methods, which are called **accessor methods**. If we take the above defined class **Box** then **@width** and **@height** are instance variables for the class Box.

```
class Box
  def initialize(w,h)
    # assign instance variables
    @width, @height = w, h
  end
end
```

## The accessor & setter Methods

To make the variables available from outside the class, they must be defined within **accessor methods**, these accessor methods are also known as a getter methods. Following example shows the usage of accessor methods –

[Live Demo](#)

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

# define a class
class Box
  # constructor method
  def initialize(w,h)
    @width, @height = w, h
  end

  # accessor methods
  def printWidth
    @width
  end

  def printHeight
    @height
  end
end

# create an object
box = Box.new(10, 20)

# use accessor methods
x = box.printWidth()
y = box.printHeight()

puts "Width of the box is : #{x}"
puts "Height of the box is : #{y}"
```

When the above code is executed, it produces the following result –

```
Width of the box is : 10
Height of the box is : 20
```

Similar to accessor methods, which are used to access the value of the variables, Ruby provides a way to set the values of those variables from outside of the class using **setter methods**, which are defined as below –

[Live Demo](#)

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

# define a class
class Box
  # constructor method
  def initialize(w,h)
    @width, @height = w, h
  end

  # accessor methods
  def getWidth
    @width
  end

  def getHeight
    @height
  end

  # setter methods
  def setWidth=(value)
    @width = value
  end
end
```

```

    def setHeight=(value)
      @height = value
    end
  end

  # create an object
  box = Box.new(10, 20)

  # use setter methods
  box.setWidth = 30
  box.setHeight = 50

  # use accessor methods
  x = box.getWidth()
  y = box.getHeight()

  puts "Width of the box is : #{x}"
  puts "Height of the box is : #{y}"

```

When the above code is executed, it produces the following result –

```

Width of the box is : 30
Height of the box is : 50

```

## The instance Methods

The **instance methods** are also defined in the same way as we define any other method using **def** keyword and they can be used using a class instance only as shown below. Their functionality is not limited to access the instance variables, but also they can do a lot more as per your requirement.

```

#!/usr/bin/ruby -w

# define a class
class Box
  # constructor method
  def initialize(w,h)
    @width, @height = w, h
  end
  # instance method
  def getArea
    @width * @height
  end
end

# create an object
box = Box.new(10, 20)

# call instance methods
a = box.getArea()
puts "Area of the box is : #{a}"

```

[Live Demo](#)

When the above code is executed, it produces the following result –

```

Area of the box is : 200

```

## The class Methods and Variables

The **class variables** is a variable, which is shared between all instances of a class. In other words, there is one instance of the variable and it is accessed by object instances. Class variables are prefixed with two @

characters (@@). A class variable must be initialized within the class definition as shown below.

A class method is defined using **def self.methodname()**, which ends with end delimiter and would be called using the class name as **classname.methodname** as shown in the following example –

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

class Box
  # Initialize our class variables
  @@count = 0
  def initialize(w,h)
    # assign instance variables
    @width, @height = w, h

    @@count += 1
  end

  def self.printCount()
    puts "Box count is : #@count"
  end
end

# create two object
box1 = Box.new(10, 20)
box2 = Box.new(30, 100)

# call class method to print box count
Box.printCount()
```

When the above code is executed, it produces the following result –

```
Box count is : 2
```

## The to\_s Method

Any class you define should have a **to\_s** instance method to return a string representation of the object. Following is a simple example to represent a Box object in terms of width and height –

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

class Box
  # constructor method
  def initialize(w,h)
    @width, @height = w, h
  end
  # define to_s method
  def to_s
    "(w:#{@width},h:#{@height})" # string formatting of the object.
  end
end

# create an object
box = Box.new(10, 20)

# to_s method will be called in reference of string automatically.
puts "String representation of box is : #{box}"
```

[Live Demo](#)

When the above code is executed, it produces the following result –

```
String representation of box is : (w:10,h:20)
```

# Access Control

Ruby gives you three levels of protection at instance methods level, which may be **public**, **private**, or **protected**. Ruby does not apply any access control over instance and class variables.

- **Public Methods** – Public methods can be called by anyone. Methods are public by default except for initialize, which is always private.
- **Private Methods** – Private methods cannot be accessed, or even viewed from outside the class. Only the class methods can access private members.
- **Protected Methods** – A protected method can be invoked only by objects of the defining class and its subclasses. Access is kept within the family.

Following is a simple example to show the syntax of all the three access modifiers –

[🔗 Live Demo](#)

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

# define a class
class Box
  # constructor method
  def initialize(w,h)
    @width, @height = w, h
  end

  # instance method by default it is public
  def getArea
    getWidth() * getHeight
  end

  # define private accessor methods
  def getWidth
    @width
  end
  def getHeight
    @height
  end
  # make them private
  private :getWidth, :getHeight

  # instance method to print area
  def printArea
    @area = getWidth() * getHeight
    puts "Big box area is : #{@area}"
  end
  # make it protected
  protected :printArea
end

# create an object
box = Box.new(10, 20)

# call instance methods
a = box.getArea()
puts "Area of the box is : #{a}"

# try to call protected or methods
box.printArea()
```

When the above code is executed, it produces the following result. Here, first method is called successfully but second method gave a problem.

```
Area of the box is : 200
test.rb:42: protected method `printArea' called for #
<Box:0xb7f11280 @height = 20, @width = 10> (NoMethodError)
```

## Class Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.

Inheritance also provides an opportunity to reuse the code functionality and fast implementation time but unfortunately Ruby does not support multiple levels of inheritances but Ruby supports **mixins**. A mixin is like a specialized implementation of multiple inheritance in which only the interface portion is inherited.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base class or superclass**, and the new class is referred to as the **derived class or sub-class**.

Ruby also supports the concept of subclassing, i.e., inheritance and following example explains the concept. The syntax for extending a class is simple. Just add a < character and the name of the superclass to your class statement. For example, following define a class *BigBox* as a subclass of *Box* –

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

# define a class
class Box
  # constructor method
  def initialize(w,h)
    @width, @height = w, h
  end
  # instance method
  def getArea
    @width * @height
  end
end

# define a subclass
class BigBox < Box

  # add a new instance method
  def printArea
    @area = @width * @height
    puts "Big box area is : #@area"
  end
end

# create an object
box = BigBox.new(10, 20)

# print the area
box.printArea()
```

[Live Demo](#)

When the above code is executed, it produces the following result –

```
Big box area is : 200
```

## Methods Overriding

Though you can add new functionality in a derived class, but sometimes you would like to change the behavior of already defined method in a parent class. You can do so simply by keeping the method name same and overriding the functionality of the method as shown below in the example –

[🔗 Live Demo](#)

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

# define a class
class Box
  # constructor method
  def initialize(w,h)
    @width, @height = w, h
  end
  # instance method
  def getArea
    @width * @height
  end
end

# define a subclass
class BigBox < Box

  # change existing getArea method as follows
  def getArea
    @area = @width * @height
    puts "Big box area is : #@area"
  end
end

# create an object
box = BigBox.new(10, 20)

# print the area using overridden method.
box.getArea()
```

## Operator Overloading

We'd like the + operator to perform vector addition of two Box objects using +, the \* operator to multiply a Box width and height by a scalar, and the unary - operator to do negate the width and height of the Box. Here is a version of the Box class with mathematical operators defined –

```
class Box
  def initialize(w,h)      # Initialize the width and height
    @width,@height = w, h
  end

  def +(other)             # Define + to do vector addition
    Box.new(@width + other.width, @height + other.height)
  end

  def -@                   # Define unary minus to negate width and height
    Box.new(-@width, -@height)
  end

  def *(scalar)            # To perform scalar multiplication
    Box.new(@width*scalar, @height*scalar)
  end
end
```



# Freezing Objects

Sometimes, we want to prevent an object from being changed. The freeze method in Object allows us to do this, effectively turning an object into a constant. Any object can be frozen by invoking **Object.freeze**. A frozen object may not be modified: you can't change its instance variables.

You can check if a given object is already frozen or not using **Object.frozen?** method, which returns true in case the object is frozen otherwise a false value is return. Following example clears the concept –

[Live Demo](#)

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

# define a class
class Box
  # constructor method
  def initialize(w,h)
    @width, @height = w, h
  end

  # accessor methods
  def getWidth
    @width
  end
  def getHeight
    @height
  end

  # setter methods
  def setWidth=(value)
    @width = value
  end
  def setHeight=(value)
    @height = value
  end
end

# create an object
box = Box.new(10, 20)

# let us freez this object
box.freeze
if( box.frozen? )
  puts "Box object is frozen object"
else
  puts "Box object is normal object"
end

# now try using setter methods
box.setWidth = 30
box.setHeight = 50

# use accessor methods
x = box.getWidth()
y = box.getHeight()

puts "Width of the box is : #{x}"
puts "Height of the box is : #{y}"
```

When the above code is executed, it produces the following result –

```
Box object is frozen object
test.rb:20:in `setWidth=': can't modify frozen object (TypeError)
```

## Class Constants

You can define a constant inside a class by assigning a direct numeric or string value to a variable, which is defined without using either `@` or `@@`. By convention, we keep constant names in upper case.

Once a constant is defined, you cannot change its value but you can access a constant directly inside a class much like a variable but if you want to access a constant outside of the class then you would have to use **classname::constant** as shown in the below example.

[Live Demo](#)

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

# define a class
class Box
  BOX_COMPANY = "TATA Inc"
  BOXWEIGHT = 10
  # constructor method
  def initialize(w,h)
    @width, @height = w, h
  end
  # instance method
  def getArea
    @width * @height
  end
end

# create an object
box = Box.new(10, 20)

# call instance methods
a = box.getArea()
puts "Area of the box is : #{a}"
puts Box::BOX_COMPANY
puts "Box weight is: #{Box::BOXWEIGHT}"
```

When the above code is executed, it produces the following result –

```
Area of the box is : 200
TATA Inc
Box weight is: 10
```

Class constants are inherited and can be overridden like instance methods.

## Create Object Using Allocate

There may be a situation when you want to create an object without calling its constructor **initialize** i.e. using **new** method, in such case you can call *allocate*, which will create an uninitialized object for you as in the following example –

[Live Demo](#)

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

# define a class
class Box
  attr_accessor :width, :height

  # constructor method
  def initialize(w,h)
    @width, @height = w, h
```

```

end

# instance method
def getArea
  @width * @height
end
end

# create an object using new
box1 = Box.new(10, 20)

# create another object using allocate
box2 = Box.allocate

# call instance method using box1
a = box1.getArea()
puts "Area of the box is : #{a}"

# call instance method using box2
a = box2.getArea()
puts "Area of the box is : #{a}"

```

When the above code is executed, it produces the following result –

```

Area of the box is : 200
test.rb:14: warning: instance variable @width not initialized
test.rb:14: warning: instance variable @height not initialized
test.rb:14:in `getArea': undefined method `*'
    for nil:NilClass (NoMethodError) from test.rb:29

```

## Class Information

If class definitions are executable code, this implies that they execute in the context of some object: self must reference something. Let's find out what it is.

```

#!/usr/bin/ruby -w

class Box
  # print class information
  puts "Type of self = #{self.type}"
  puts "Name of self = #{self.name}"
end

```

When the above code is executed, it produces the following result –

```

Type of self = Class
Name of self = Box

```

This means that a class definition is executed with that class as the current object. This means that methods in the metaclass and its superclasses will be available during the execution of the method definition.

## 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

[Live Demo](#)

```
#!/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 –

Sr.No.	Modifier & Description
1	<b>i</b> Ignores case when matching text.
2	<b>o</b> Performs <code>#{} </code> interpolations only once, the first time the regexp literal is evaluated.
3	<b>x</b> Ignores whitespace and allows comments in regular expressions.
4	<b>m</b> Matches multiple lines, recognizing newlines as normal characters.
5	<b>u,e,s,n</b> Interprets 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.

## Regular-Expression Examples

<b>Literal Characters</b>	▶
<b>Character Classes</b>	▶
<b>Special Character Classes</b>	▶
<b>Repetition Cases</b>	▶
<b>Non-greedy Repetition</b>	▶
<b>Grouping with Parentheses</b>	▶
<b>Back References</b>	▶
<b>Alternatives</b>	▶
<b>Anchors</b>	▶
<b>Special Syntax with Parentheses</b>	▶

## 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!** replaces the first occurrence of the pattern and **gsub** & **gsub!** replaces all occurrences.

The **sub** and **gsub** returns a new string, leaving the original unmodified where as **sub!** and **gsub!** modify the string on which they are called.

Following is the example –

[Live Demo](#)

```
#!/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 –

[Live Demo](#)

```
#!/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/DBI Tutorial

This chapter teaches you how to access a database using Ruby. The *Ruby DBI* module provides a database-independent interface for Ruby scripts similar to that of the Perl DBI module.

DBI stands for Database Independent Interface for Ruby, which means DBI provides an abstraction layer between the Ruby code and the underlying database, allowing you to switch database implementations really easily. It defines a set of methods, variables, and conventions that provide a consistent database interface, independent of the actual database being used.

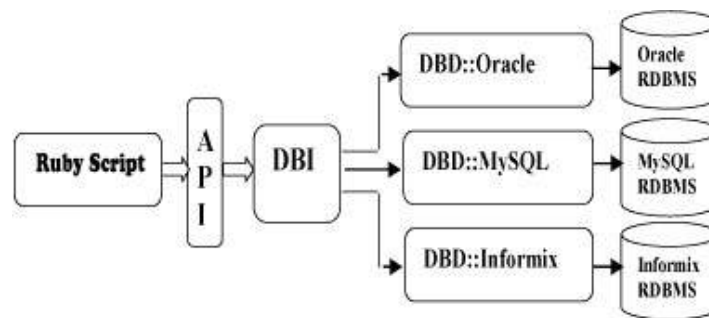
DBI can interface with the following –

- ADO (ActiveX Data Objects)

- ▣ DB2
- ▣ Frontbase
- ▣ mSQL
- ▣ MySQL
- ▣ ODBC
- ▣ Oracle
- ▣ OCI8 (Oracle)
- ▣ PostgreSQL
- ▣ Proxy/Server
- ▣ SQLite
- ▣ SQLRelay

## Architecture of a DBI Application

DBI is independent of any database available in the backend. You can use DBI whether you are working with Oracle, MySQL or Informix, etc. This is clear from the following architecture diagram.




The general architecture for Ruby DBI uses two layers –

- ▣ The database interface (DBI) layer. This layer is database independent and provides a set of common access methods that are used the same way regardless of the type of database server with which you're communicating.
- ▣ The database driver (DBD) layer. This layer is database dependent; different drivers provide access to different database engines. There is one driver for MySQL, another for PostgreSQL, another for InterBase, another for Oracle, and so forth. Each driver interprets requests from the DBI layer and maps them onto requests appropriate for a given type of database server.

## Prerequisites

If you want to write Ruby scripts to access MySQL databases, you'll need to have the Ruby MySQL module installed.

This module acts as a DBD as explained above and can be downloaded from <https://www.tmtm.org/en/mysql/ruby/> 

## Obtaining and Installing Ruby/DBI

You can download and install the Ruby DBI module from the following location –

<https://imgur.com/NFEuWe4/embed> 

Before starting this installation make sure you have the root privilege. Now, follow the steps given below –

## Step 1

```
$ tar xzf dbi-0.2.0.tar.gz
```

## Step 2

Go in distribution directory *dbi-0.2.0* and configure it using the *setup.rb* script in that directory. The most general configuration command looks like this, with no arguments following the config argument. This command configures the distribution to install all drivers by default.

```
$ ruby setup.rb config
```

To be more specific, provide a *--with* option that lists the particular parts of the distribution you want to use. For example, to configure only the main DBI module and the MySQL DBD-level driver, issue the following command –

```
$ ruby setup.rb config --with = dbi,dbd_mysql
```

## Step 3

Final step is to build the driver and install it using the following commands –

```
$ ruby setup.rb setup  
$ ruby setup.rb install
```

## Database Connection

Assuming we are going to work with MySQL database, before connecting to a database make sure of the following –

- You have created a database TESTDB.
- You have created EMPLOYEE in TESTDB.
- This table is having fields FIRST\_NAME, LAST\_NAME, AGE, SEX, and INCOME.
- User ID "testuser" and password "test123" are set to access TESTDB.
- Ruby Module DBI is installed properly on your machine.
- You have gone through MySQL tutorial to understand MySQL Basics.

Following is the example of connecting with MySQL database "TESTDB"

```
#!/usr/bin/ruby -w  
  
require "dbi"  
  
begin  
  # connect to the MySQL server  
  dbh = DBI.connect("DBI:MySQL:TESTDB:localhost", "testuser", "test123")
```



```

# get server version string and display it
row = dbh.select_one("SELECT VERSION()")
puts "Server version: " + row[0]
rescue DBI::DatabaseError => e
  puts "An error occurred"
  puts "Error code:      #{e.err}"
  puts "Error message:  #{e.errstr}"
ensure
  # disconnect from server
  dbh.disconnect if dbh
end

```

While running this script, it produces the following result at our Linux machine.

```
Server version: 5.0.45
```

If a connection is established with the data source, then a Database Handle is returned and saved into **dbh** for further use otherwise **dbh** is set to nil value and **e.err** and **e.errstr** return error code and an error string respectively.

Finally, before coming out it, ensure that database connection is closed and resources are released.

## INSERT Operation

INSERT operation is required when you want to create your records into a database table.

Once a database connection is established, we are ready to create tables or records into the database tables using **do** method or **prepare** and **execute** method.

## Using do Statement

Statements that do not return rows can be issued by invoking the **do** database handle method. This method takes a statement string argument and returns a count of the number of rows affected by the statement.

```

dbh.do("DROP TABLE IF EXISTS EMPLOYEE")
dbh.do("CREATE TABLE EMPLOYEE (
  FIRST_NAME  CHAR(20) NOT NULL,
  LAST_NAME   CHAR(20),
  AGE INT,
  SEX CHAR(1),
  INCOME FLOAT )" );

```

Similarly, you can execute the SQL *INSERT* statement to create a record into the EMPLOYEE table.

```

#!/usr/bin/ruby -w

require "dbi"

begin
  # connect to the MySQL server
  dbh = DBI.connect("DBI:Mysql:TESTDB:localhost", "testuser", "test123")
  dbh.do( "INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)
    VALUES ('Mac', 'Mohan', 20, 'M', 2000)" )
  puts "Record has been created"
  dbh.commit
rescue DBI::DatabaseError => e
  puts "An error occurred"
  puts "Error code:      #{e.err}"
  puts "Error message:  #{e.errstr}"
  dbh.rollback
end

```

```
ensure
  # disconnect from server
  dbh.disconnect if dbh
end
```

## Using prepare and execute

You can use *prepare* and *execute* methods of DBI class to execute the SQL statement through Ruby code.

Record creation takes the following steps –

- Preparing SQL statement with INSERT statement. This will be done using the **prepare** method.
- Executing SQL query to select all the results from the database. This will be done using the **execute** method.
- Releasing Statement handle. This will be done using **finish** API
- If everything goes fine, then **commit** this operation otherwise you can **rollback** the complete transaction.

Following is the syntax to use these two methods –

```
sth = dbh.prepare(statement)
sth.execute
... zero or more SQL operations ...
sth.finish
```

These two methods can be used to pass **bind** values to SQL statements. There may be a case when values to be entered is not given in advance. In such a case, binding values are used. A question mark (?) is used in place of actual values and then actual values are passed through execute() API.

Following is the example to create two records in the EMPLOYEE table –

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

require "dbi"

begin
  # connect to the MySQL server
  dbh = DBI.connect("DBI:Mysql:TESTDB:localhost", "testuser", "test123")
  sth = dbh.prepare( "INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)
    VALUES (?, ?, ?, ?, ?)" )
  sth.execute('John', 'Poul', 25, 'M', 2300)
  sth.execute('Zara', 'Ali', 17, 'F', 1000)
  sth.finish
  dbh.commit
  puts "Record has been created"
rescue DBI::DatabaseError => e
  puts "An error occurred"
  puts "Error code:      #{e.err}"
  puts "Error message: #{e.errstr}"
  dbh.rollback
ensure
  # disconnect from server
  dbh.disconnect if dbh
end
```

If there are multiple INSERTs at a time, then preparing a statement first and then executing it multiple times within a loop is more efficient than invoking do each time through the loop.

## READ Operation

READ Operation on any database means to fetch some useful information from the database.

Once our database connection is established, we are ready to make a query into this database. We can use either **do** method or **prepare** and **execute** methods to fetch values from a database table.

Record fetching takes following steps –

- Preparing SQL query based on required conditions. This will be done using the **prepare** method.
- Executing SQL query to select all the results from the database. This will be done using the **execute** method.
- Fetching all the results one by one and printing those results. This will be done using the **fetch** method.
- Releasing Statement handle. This will be done using the **finish** method.

Following is the procedure to query all the records from EMPLOYEE table having salary more than 1000.

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

require "dbi"

begin
  # connect to the MySQL server
  dbh = DBI.connect("DBI:Mysql:TESTDB:localhost", "testuser", "test123")
  sth = dbh.prepare("SELECT * FROM EMPLOYEE WHERE INCOME > ?")
  sth.execute(1000)

  sth.fetch do |row|
    printf "First Name: %s, Last Name : %s\n", row[0], row[1]
    printf "Age: %d, Sex : %s\n", row[2], row[3]
    printf "Salary :%d \n\n", row[4]
  end
  sth.finish
rescue DBI::DatabaseError => e
  puts "An error occurred"
  puts "Error code:      #{e.err}"
  puts "Error message: #{e.errstr}"
ensure
  # disconnect from server
  dbh.disconnect if dbh
end
```

This will produce the following result –

```
First Name: Mac, Last Name : Mohan
Age: 20, Sex : M
Salary :2000
```

```
First Name: John, Last Name : Poul
Age: 25, Sex : M
Salary :2300
```

There are more short cut methods to fetch records from the database. If you are interested then go through the Fetching the Result [↗](#) otherwise proceed to the next section.

## Update Operation

UPDATE Operation on any database means to update one or more records, which are already available in the database. Following is the procedure to update all the records having SEX as 'M'. Here, we will increase AGE of all the males by one year. This will take three steps –

- ▣ Preparing SQL query based on required conditions. This will be done using the **prepare** method.
- ▣ Executing SQL query to select all the results from the database. This will be done using the **execute** method.
- ▣ Releasing Statement handle. This will be done using the **finish** method.
- ▣ If everything goes fine then **commit** this operation otherwise you can **rollback** the complete transaction.

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

require "dbi"

begin
  # connect to the MySQL server
  dbh = DBI.connect("DBI:Mysql:TESTDB:localhost", "testuser", "test123")
  sth = dbh.prepare("UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = ?")
  sth.execute('M')
  sth.finish
  dbh.commit
rescue DBI::DatabaseError => e
  puts "An error occurred"
  puts "Error code:      #{e.err}"
  puts "Error message:  #{e.errstr}"
  dbh.rollback
ensure
  # disconnect from server
  dbh.disconnect if dbh
end
```

## DELETE Operation

DELETE operation is required when you want to delete some records from your database. Following is the procedure to delete all the records from EMPLOYEE where AGE is more than 20. This operation will take following steps.

- ▣ Preparing SQL query based on required conditions. This will be done using the **prepare** method.
- ▣ Executing SQL query to delete required records from the database. This will be done using the **execute** method.
- ▣ Releasing Statement handle. This will be done using the **finish** method.
- ▣ If everything goes fine then **commit** this operation otherwise you can **rollback** the complete transaction.

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

require "dbi"
```

```

begin
  # connect to the MySQL server
  dbh = DBI.connect("DBI:Mysql:TESTDB:localhost", "testuser", "test123")
  sth = dbh.prepare("DELETE FROM EMPLOYEE WHERE AGE > ?")
  sth.execute(20)
  sth.finish
  dbh.commit
rescue DBI::DatabaseError => e
  puts "An error occurred"
  puts "Error code:      #{e.err}"
  puts "Error message: #{e.errstr}"
  dbh.rollback
ensure
  # disconnect from server
  dbh.disconnect if dbh
end

```

## Performing Transactions

Transactions are a mechanism that ensures data consistency. Transactions should have the following four properties –

- **Atomicity** – Either a transaction completes or nothing happens at all.
- **Consistency** – A transaction must start in a consistent state and leave the system in a consistent state.
- **Isolation** – Intermediate results of a transaction are not visible outside the current transaction.
- **Durability** – Once a transaction was committed, the effects are persistent, even after a system failure.

The DBI provides two methods to either *commit* or *rollback* a transaction. There is one more method called *transaction* which can be used to implement transactions. There are two simple approaches to implement transactions –

### Approach I

The first approach uses DBI's *commit* and *rollback* methods to explicitly commit or cancel the transaction –

```

dbh['AutoCommit'] = false # Set auto commit to false.
begin
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1 WHERE FIRST_NAME = 'John'")
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1 WHERE FIRST_NAME = 'Zara'")
  dbh.commit
rescue
  puts "transaction failed"
  dbh.rollback
end
dbh['AutoCommit'] = true

```

### Approach II

The second approach uses the *transaction* method. This is simpler, because it takes a code block containing the statements that make up the transaction. The *transaction* method executes the block, then invokes *commit* or *rollback* automatically, depending on whether the block succeeds or fails –

```

dbh['AutoCommit'] = false # Set auto commit to false.
dbh.transaction do |dbh|
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1 WHERE FIRST_NAME = 'John'")
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1 WHERE FIRST_NAME = 'Zara'")
end

```

```
end
dbh['AutoCommit'] = true
```

## COMMIT Operation

Commit is the operation, which gives a green signal to database to finalize the changes, and after this operation, no change can be reverted back.

Here is a simple example to call the **commit** method.

```
dbh.commit
```

## ROLLBACK Operation

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use the **rollback** method.

Here is a simple example to call the **rollback** method.

```
dbh.rollback
```

## Disconnecting Database

To disconnect Database connection, use disconnect API.

```
dbh.disconnect
```

If the connection to a database is closed by the user with the disconnect method, any outstanding transactions are rolled back by the DBI. However, instead of depending on any of DBI's implementation details, your application would be better off calling the commit or rollback explicitly.

## Handling Errors

There are many sources of errors. A few examples are a syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already canceled or finished statement handle.

If a DBI method fails, DBI raises an exception. DBI methods may raise any of several types of exception but the two most important exception classes are *DBI::InterfaceError* and *DBI::DatabaseError*.

Exception objects of these classes have three attributes named *err*, *errstr*, and *state*, which represent the error number, a descriptive error string, and a standard error code. The attributes are explained below –

- **err** – Returns an integer representation of the occurred error or *nil* if this is not supported by the DBD. The Oracle DBD for example returns the numerical part of an *ORA-XXXX* error message.
- **errstr** – Returns a string representation of the occurred error.
- **state** – Returns the SQLSTATE code of the occurred error. The SQLSTATE is a five-character-long string. Most DBDs do not support this and return *nil* instead.

You have seen following code above in most of the examples –

```

rescue DBI::DatabaseError => e
  puts "An error occurred"
  puts "Error code:      #{e.err}"
  puts "Error message:  #{e.errstr}"
  dbh.rollback
ensure
  # disconnect from server
  dbh.disconnect if dbh
end

```

To get debugging information about what your script is doing as it executes, you can enable tracing. To do this, you must first load the `dbi/trace` module and then call the `trace` method that controls the trace mode and output destination –

```

require "dbi/trace"
.....

trace(mode, destination)

```

The mode value may be 0 (off), 1, 2, or 3, and the destination should be an IO object. The default values are 2 and `STDERR`, respectively.

## Code Blocks with Methods

There are some methods that create handles. These methods can be invoked with a code block. The advantage of using code block along with methods is that they provide the handle to the code block as its parameter and automatically cleans up the handle when the block terminates. There are few examples to understand the concept.

- **DBI.connect** – This method generates a database handle and it is recommended to call *disconnect* at the end of the block to disconnect the database.
- **dbh.prepare** – This method generates a statement handle and it is recommended to *finish* at the end of the block. Within the block, you must invoke *execute* method to execute the statement.
- **dbh.execute** – This method is similar except we don't need to invoke *execute* within the block. The statement handle is automatically executed.

### Example 1

**DBI.connect** can take a code block, passes the database handle to it, and automatically disconnects the handle at the end of the block as follows.

```
dbh = DBI.connect("DBI:Mysql:TESTDB:localhost", "testuser", "test123") do |dbh|
```

### Example 2

**dbh.prepare** can take a code block, passes the statement handle to it, and automatically calls *finish* at the end of the block as follows.

```

dbh.prepare("SHOW DATABASES") do |sth|
  sth.execute
  puts "Databases: " + sth.fetch_all.join(", ")
end

```

### Example 3

**dbh.execute** can take a code block, passes the statement handle to it, and automatically calls finish at the end of the block as follows –

```
dbh.execute("SHOW DATABASES") do |sth|
  puts "Databases: " + sth.fetch_all.join(", ")
end
```

DBI *transaction* method also takes a code block which has been described in above.

## Driver-specific Functions and Attributes

The DBI lets the database drivers provide additional database-specific functions, which can be called by the user through the *func* method of any Handle object.

Driver-specific attributes are supported and can be set or gotten using the `[]=` or `[]` methods.

DBD::Mysql implements the following driver-specific functions –

### Example

```
#!/usr/bin/ruby

require "dbi"
begin
  # connect to the MySQL server
  dbh = DBI.connect("DBI:Mysql:TESTDB:localhost", "testuser", "test123")
  puts dbh.func(:client_info)
  puts dbh.func(:client_version)
  puts dbh.func(:host_info)
  puts dbh.func(:proto_info)
  puts dbh.func(:server_info)
  puts dbh.func(:thread_id)
  puts dbh.func(:stat)
rescue DBI::DatabaseError => e
  puts "An error occurred"
  puts "Error code:      #{e.err}"
  puts "Error message:  #{e.errstr}"
ensure
  dbh.disconnect if dbh
end
```

This will produce the following result –

```
5.0.45
50045
Localhost via UNIX socket
10
5.0.45
150621
Uptime: 384981  Threads: 1  Questions: 1101078  Slow queries: 4 \
Opens: 324  Flush tables: 1  Open tables: 64 \
Queries per second avg: 2.860
```



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 –

[Live Demo](#)

```
#!/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 <https://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 <https://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 –

[Live Demo](#)

```
#!/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 the 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>
```

## Quoting Strings

When dealing with URLs and HTML code, you must be careful to quote certain characters. For instance, a slash character ( / ) has special meaning in a URL, so it must be **escaped** if it's not part of the pathname.

For example, any / in the query portion of the URL will be translated to the string %2F and must be translated back to a / for you to use it. Space and ampersand are also special characters. To handle this, CGI provides the

routines **CGI.escape** and **CGI.unescape**.

```
#!/usr/bin/ruby

require 'cgi'
puts CGI.escape("Zara Ali/A Sweet & Sour Girl")
```

This will produce the following result –

```
Zara+Ali%2FA+Sweet+%26+Sour+Girl")
```

```
#!/usr/bin/ruby

require 'cgi'
puts CGI.escapeHTML('<h1>Zara Ali/A Sweet & Sour Girl</h1>')
```

This will produce the following result –

```
&lt;h1&gt;Zara Ali/A Sweet & Sour Girl&lt;/h1&gt;'
```

## Useful Methods in CGI Class

Here is the list of methods related to CGI class –

- The Ruby CGI [↗](#) – Methods related to Standard CGI library.

## Cookies and Sessions

We have explained these two concepts in different sections. Please follow the sections –

- The Ruby CGI Cookies [↗](#) – How to handle CGI Cookies.
- The Ruby CGI Sessions [↗](#) – How to manage CGI sessions.

## Web Hosting Servers

You could check the following topic on the internet to host your website on a Unix-based Server –

- Unix-based Web hosting [↗](#)

## Sending Email using Ruby - SMTP

Simple Mail Transfer Protocol (SMTP) is a protocol, which handles sending e-mail and routing e-mail between mail servers.

Ruby provides Net::SMTP class for Simple Mail Transfer Protocol (SMTP) client-side connection and provides two class methods *new* and *start*.

- The **new** takes two parameters –
  - The *server name* defaulting to localhost.
  - The *port number* defaulting to the well-known port 25.
- The **start** method takes these parameters –

- The *server* – IP name of the SMTP server, defaulting to localhost.
- The *port* – Port number, defaulting to 25.
- The *domain* – Domain of the mail sender, defaulting to ENV["HOSTNAME"].
- The *account* – Username, default is nil.
- The *password* – User password, defaulting to nil.
- The *authtype* – Authorization type, defaulting to *cram\_md5*.

An SMTP object has an instance method called `sendmail`, which will typically be used to do the work of mailing a message. It takes three parameters –

- The *source* – A string or array or anything with an *each* iterator returning one string at a time.
- The *sender* – A string that will appear in the *from* field of the email.
- The *recipients* – A string or an array of strings representing the recipients' addressee(s).

## Example

Here is a simple way to send one email using Ruby script. Try it once –

```
require 'net/smtp'

message = <<MESSAGE_END
From: Private Person <me@fromdomain.com>
To: A Test User <test@todomain.com>
Subject: SMTP e-mail test

This is a test e-mail message.
MESSAGE_END

Net::SMTP.start('localhost') do |smtp|
  smtp.send_message message, 'me@fromdomain.com', 'test@todomain.com'
end
```

Here, you have placed a basic e-mail in message, using a document, taking care to format the headers correctly. E-mails require a **From**, **To**, and **Subject** header, separated from the body of the e-mail with a blank line.

To send the mail you use `Net::SMTP` to connect to the SMTP server on the local machine and then use the `send_message` method along with the message, the from address, and the destination address as parameters (even though the from and to addresses are within the e-mail itself, these aren't always used to route mail).

If you're not running an SMTP server on your machine, you can use the `Net::SMTP` to communicate with a remote SMTP server. Unless you're using a webmail service (such as Hotmail or Yahoo! Mail), your e-mail provider will have provided you with outgoing mail server details that you can supply to `Net::SMTP`, as follows –

```
Net::SMTP.start('mail.your-domain.com')
```

This line of code connects to the SMTP server on port 25 of `mail.your-domain.com` without using any username or password. If you need to, though, you can specify port number and other details. For example –

```
Net::SMTP.start('mail.your-domain.com',
  25,
```

```
'localhost',  
'username', 'password' :plain)
```

This example connects to the SMTP server at mail.your-domain.com using a username and password in plain text format. It identifies the client's hostname as localhost.

## Sending an HTML e-mail using Ruby

When you send a text message using Ruby then all the content will be treated as simple text. Even if you will include HTML tags in a text message, it will be displayed as simple text and HTML tags will not be formatted according to HTML syntax. But Ruby Net::SMTP provides option to send an HTML message as actual HTML message.

While sending an email message you can specify a Mime version, content type and character set to send an HTML email.

### Example

Following is the example to send HTML content as an email. Try it once –

```
require 'net/smtp'  
  
message = <<MESSAGE_END  
From: Private Person <me@fromdomain.com>  
To: A Test User <test@todomain.com>  
MIME-Version: 1.0  
Content-type: text/html  
Subject: SMTP e-mail test  
  
This is an e-mail message to be sent in HTML format  
  
<b>This is HTML message.</b>  
<h1>This is headline.</h1>  
MESSAGE_END  
  
Net::SMTP.start('localhost') do |smtp|  
  smtp.send_message message, 'me@fromdomain.com', 'test@todomain.com'  
end
```

## Sending Attachments as an e-mail

To send an email with mixed content requires to set **Content-type** header to **multipart/mixed**. Then text and attachment sections can be specified within **boundaries**.

A boundary is started with two hyphens followed by a unique number, which cannot appear in the message part of the email. A final boundary denoting the email's final section must also end with two hyphens.

Attached files should be encoded with the **pack("m")** function to have base64 encoding before transmission.

### Example

Following is the example, which will send a file **/tmp/test.txt** as an attachment.

```
require 'net/smtp'  
  
filename = "/tmp/test.txt"  
# Read a file and encode it into base64 format  
filecontent = File.read(filename)
```

```

encodedcontent = [filecontent].pack("m") # base64

marker = "AUNIQUEMARKER"
body = <<EOF
This is a test email to send an attachement.
EOF

# Define the main headers.
part1 = <<EOF
From: Private Person <me@fromdomain.net>
To: A Test User <test@todmain.com>
Subject: Sending Attachement
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary = #{marker}
--#{marker}
EOF

# Define the message action
part2 = <<EOF
Content-Type: text/plain
Content-Transfer-Encoding:8bit

#{body}
--#{marker}
EOF

# Define the attachment section
part3 = <<EOF
Content-Type: multipart/mixed; name = \"#{filename}\"
Content-Transfer-Encoding:base64
Content-Disposition: attachment; filename = \"#{filename}\"

#{encodedcontent}
--#{marker}--
EOF

mailtext = part1 + part2 + part3

# Let's put our code in safe area
begin
  Net::SMTP.start('localhost') do |smtp|
    smtp.sendmail(mailtext, 'me@fromdomain.net', ['test@todmain.com'])
  end
rescue Exception => e
  print "Exception occured: " + e
end

```

**NOTE** – You can specify multiple destinations inside the array but they should be separated by comma.

## Ruby - Socket Programming

Ruby provides two levels of access to network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

Ruby also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

This chapter gives you an understanding on most famous concept in Networking – Socket Programming.

### What are Sockets?

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The *socket* provides specific classes for handling the common transports as well as a generic interface for handling the rest.

Sockets have their own vocabulary –

Sr.No.	Term & Description
1	<b>domain</b> The family of protocols that will be used as the transport mechanism. These values are constants such as PF_INET, PF_UNIX, PF_X25, and so on.
2	<b>type</b> The type of communications between the two endpoints, typically SOCK_STREAM for connection-oriented protocols and SOCK_DGRAM for connectionless protocols.
3	<b>protocol</b> Typically zero, this may be used to identify a variant of a protocol within a domain and type.
4	<b>hostname</b> The identifier of a network interface –  A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation  A string "<broadcast>", which specifies an INADDR_BROADCAST address.  A zero-length string, which specifies INADDR_ANY, or  An Integer, interpreted as a binary address in host byte order.
5	<b>port</b> Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service.

## A Simple Client

Here we will write a very simple client program, which will open a connection to a given port and given host. Ruby class **TCPSocket** provides *open* function to open such a socket.

The **TCPSocket.open(hostname, port )** opens a TCP connection to *hostname* on the *port*.

Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.



The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits –

```
require 'socket'          # Sockets are in standard library

hostname = 'localhost'
port = 2000

s = TCPSocket.open(hostname, port)

while line = s.gets      # Read lines from the socket
  puts line.chop         # And print with platform line terminator
end
s.close                  # Close the socket when done
```

## A Simple Server

To write Internet servers, we use the **TCPServer** class. A TCPServer object is a factory for TCPSocket objects.

Now call **TCPServer.open(hostname, port)** function to specify a *port* for your service and create a **TCPServer** object.

Next, call the *accept* method of the returned TCPServer object. This method waits until a client connects to the port you specified, and then returns a *TCPSocket* object that represents the connection to that client.

```
require 'socket'          # Get sockets from stdlib

server = TCPServer.open(2000) # Socket to listen on port 2000
loop {                      # Servers run forever
  client = server.accept      # Wait for a client to connect
  client.puts(Time.now.ctime) # Send the time to the client
  client.puts "Closing the connection. Bye!"
  client.close                # Disconnect from the client
}
```

Now, run this server in background and then run the above client to see the result.

## Multi-Client TCP Servers

Most servers on the Internet are designed to deal with large numbers of clients at any one time.

Ruby's *Thread* class makes it easy to create a multithreaded server.one that accepts requests and immediately creates a new thread of execution to process the connection while allowing the main program to await more connections –

```
require 'socket'          # Get sockets from stdlib

server = TCPServer.open(2000) # Socket to listen on port 2000
loop {                      # Servers run forever
  Thread.start(server.accept) do |client|
    client.puts(Time.now.ctime) # Send the time to the client
    client.puts "Closing the connection. Bye!"
    client.close                # Disconnect from the client
  end
}
```

In this example, you have a permanent loop, and when `server.accept` responds, a new thread is created and started immediately to handle the connection that has just been accepted, using the connection object passed

into the thread. However, the main program immediately loops back and awaits new connections.

Using Ruby threads in this way means the code is portable and will run in the same way on Linux, OS X, and Windows.

## A Tiny Web Browser

We can use the socket library to implement any Internet protocol. Here, for example, is a code to fetch the content of a web page –

```
require 'socket'

host = 'www.tutorialspoint.com'    # The web server
port = 80                          # Default HTTP port
path = "/index.htm"               # The file we want

# This is the HTTP request we send to fetch a file
request = "GET #{path} HTTP/1.0\r\n\r\n"

socket = TCPSocket.open(host,port) # Connect to server
socket.print(request)               # Send request
response = socket.read              # Read complete response
# Split response at first blank line into headers and body
headers,body = response.split("\r\n\r\n", 2)
print body                          # And display it
```

To implement the similar web client, you can use a pre-built library like **Net::HTTP** for working with HTTP. Here is the code that does the equivalent of the previous code –

```
require 'net/http'                  # The library we need
host = 'www.tutorialspoint.com'    # The web server
path = '/index.htm'                # The file we want

http = Net::HTTP.new(host)          # Create a connection
headers, body = http.get(path)      # Request the file
if headers.code == "200"            # Check the status code
  print body
else
  puts "#{headers.code} #{headers.message}"
end
```

Please check similar libraries to work with FTP, SMTP, POP, and IMAP protocols.

## Further Readings

We have given you a quick start on Socket Programming. It is a big subject, so it is recommended that you go through Ruby Socket Library and Class Methods [↗](#) to find more details.

## Ruby - XML, XSLT and XPath Tutorial

### What is XML?

The Extensible Markup Language (XML) is a markup language much like HTML or SGML. This is recommended by the World Wide Web Consortium and available as an open standard.

XML is a portable, open source language that allows programmers to develop applications that can be read by other applications, regardless of operating system and/or developmental language.

XML is extremely useful for keeping track of small to medium amounts of data without requiring a SQL-based backbone.

## XML Parser Architectures and APIs

There are two different flavors available for XML parsers –

- **SAX-like (Stream interfaces)** – Here you register callbacks for events of interest and then let the parser proceed through the document. This is useful when your documents are large or you have memory limitations, it parses the file as it reads it from disk, and the entire file is never stored in memory.
- **DOM-like (Object tree interfaces)** – This is World Wide Web Consortium recommendation wherein the entire file is read into memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document.

SAX obviously can't process information as fast as DOM can when working with large files. On the other hand, using DOM exclusively can really kill your resources, especially if used on a lot of small files.

SAX is read-only, while DOM allows changes to the XML file. Since these two different APIs literally complement each other there is no reason why you can't use them both for large projects.

## Parsing and Creating XML using Ruby

The most common way to manipulate XML is with the REXML library by Sean Russell. Since 2002, REXML has been part of the standard Ruby distribution.

REXML is a pure-Ruby XML processor conforming to the XML 1.0 standard. It is a *non-validating* processor, passing all of the OASIS non-validating conformance tests.

REXML parser has the following advantages over other available parsers –

- It is written 100 percent in Ruby.
- It can be used for both SAX and DOM parsing.
- It is lightweight, less than 2000 lines of code.
- Methods and classes are really easy-to-understand.
- SAX2-based API and Full XPath support.
- Shipped with Ruby installation and no separate installation is required.

For all our XML code examples, let's use a simple XML file as an input –

```
<collection shelf = "New Arrivals">
  <movie title = "Enemy Behind">
    <type>War, Thriller</type>
    <format>DVD</format>
    <year>2003</year>
    <rating>PG</rating>
    <stars>10</stars>
    <description>Talk about a US-Japan war</description>
  </movie>
  <movie title = "Transformers">
    <type>Anime, Science Fiction</type>
    <format>DVD</format>
    <year>1989</year>
```

```

    <rating>R</rating>
    <stars>8</stars>
    <description>A schientific fiction</description>
  </movie>
  <movie title = "Trigun">
    <type>Anime, Action</type>
    <format>DVD</format>
    <episodes>4</episodes>
    <rating>PG</rating>
    <stars>10</stars>
    <description>Vash the Stampede!</description>
  </movie>
  <movie title = "Ishtar">
    <type>Comedy</type>
    <format>VHS</format>
    <rating>PG</rating>
    <stars>2</stars>
    <description>Viewable boredom</description>
  </movie>
</collection>

```

## DOM-like Parsing

Let's first parse our XML data in *tree fashion*. We begin by requiring the **rexml/document** library; often we do an include REXML to import into the top-level namespace for convenience.

```

#!/usr/bin/ruby -w

require 'rexml/document'
include REXML

xmlfile = File.new("movies.xml")
xmldoc = Document.new(xmlfile)

# Now get the root element
root = xmldoc.root
puts "Root element : " + root.attributes["shelf"]

# This will output all the movie titles.
xmldoc.elements.each("collection/movie"){
  |e| puts "Movie Title : " + e.attributes["title"]
}

# This will output all the movie types.
xmldoc.elements.each("collection/movie/type") {
  |e| puts "Movie Type : " + e.text
}

# This will output all the movie description.
xmldoc.elements.each("collection/movie/description") {
  |e| puts "Movie Description : " + e.text
}

```

This will produce the following result –

```

Root element : New Arrivals
Movie Title : Enemy Behind
Movie Title : Transformers
Movie Title : Trigun
Movie Title : Ishtar
Movie Type : War, Thriller

```

```
Movie Type : Anime, Science Fiction
Movie Type : Anime, Action
Movie Type : Comedy
Movie Description : Talk about a US-Japan war
Movie Description : A schientific fiction
Movie Description : Vash the Stampede!
Movie Description : Viewable boredom
```

## SAX-like Parsing

To process the same data, *movies.xml*, file in a *stream-oriented* way we will define a *listener* class whose methods will be the target of *callbacks* from the parser.

**NOTE** – It is not suggested to use SAX-like parsing for a small file, this is just for a demo example.

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

require 'rexml/document'
require 'rexml/streamlistener'
include REXML

class MyListener
  include REXML::StreamListener
  def tag_start(*args)
    puts "tag_start: #{args.map {|x| x.inspect}.join(', ')}"
  end

  def text(data)
    return if data =~ /\s$/ # whitespace only
    abbrev = data[0..40] + (data.length > 40 ? "..." : "")
    puts "  text    :  #{abbrev.inspect}"
  end
end

list = MyListener.new
xmlfile = File.new("movies.xml")
Document.parse_stream(xmlfile, list)
```

This will produce the following result –

```
tag_start: "collection", {"shelf"=>"New Arrivals"}
tag_start: "movie", {"title"=>"Enemy Behind"}
tag_start: "type", {}
  text    :  "War, Thriller"
tag_start: "format", {}
tag_start: "year", {}
tag_start: "rating", {}
tag_start: "stars", {}
tag_start: "description", {}
  text    :  "Talk about a US-Japan war"
tag_start: "movie", {"title"=>"Transformers"}
tag_start: "type", {}
  text    :  "Anime, Science Fiction"
tag_start: "format", {}
tag_start: "year", {}
```

```

tag_start: "rating", {}
tag_start: "stars", {}
tag_start: "description", {}
  text    :    "A schientific fiction"
tag_start: "movie", {"title"=>"Trigun"}
tag_start: "type", {}
  text    :    "Anime, Action"
tag_start: "format", {}
tag_start: "episodes", {}
tag_start: "rating", {}
tag_start: "stars", {}
tag_start: "description", {}
  text    :    "Vash the Stampede!"
tag_start: "movie", {"title"=>"Ishtar"}
tag_start: "type", {}
tag_start: "format", {}
tag_start: "rating", {}
tag_start: "stars", {}
tag_start: "description", {}
  text    :    "Viewable boredom"

```

## XPath and Ruby

An alternative way to view XML is XPath. This is a kind of pseudo-language that describes how to locate specific elements and attributes in an XML document, treating that document as a logical ordered tree.

REXML has XPath support via the *XPath* class. It assumes tree-based parsing (document object model) as we have seen above.

```

#!/usr/bin/ruby -w

require 'rexml/document'
include REXML

xmlfile = File.new("movies.xml")
xmldoc = Document.new(xmlfile)

# Info for the first movie found
movie = XPath.first(xmldoc, "//movie")
p movie

# Print out all the movie types
XPath.each(xmldoc, "//type") { |e| puts e.text }

# Get an array of all of the movie formats.
names = XPath.match(xmldoc, "//format").map {|x| x.text }
p names

```

This will produce the following result –

```

<movie title = 'Enemy Behind'> ... </>
War, Thriller
Anime, Science Fiction
Anime, Action

```

## XSLT and Ruby

There are two XSLT parsers available that Ruby can use. A brief description of each is given here.

### Ruby-Sablotron

This parser is written and maintained by Masayoshi Takahashi. This is written primarily for Linux OS and requires the following libraries –

- ▣ Sablot
- ▣ Iconv
- ▣ Expat

You can find this module at **Ruby-Sablotron**.

### XSLT4R

XSLT4R is written by Michael Neumann and can be found at the RAA in the Library section under XML. XSLT4R uses a simple commandline interface, though it can alternatively be used within a third-party application to transform an XML document.

XSLT4R needs XMLScan to operate, which is included within the XSLT4R archive and which is also a 100 percent Ruby module. These modules can be installed using standard Ruby installation method (i.e., ruby install.rb).

XSLT4R has the following syntax –

```
ruby xslt.rb stylesheet.xml document.xml [arguments]
```

If you want to use XSLT4R from within an application, you can include XSLT and input the parameters you need. Here is the example –

```
require "xslt"

stylesheet = File.readlines("stylesheet.xml").to_s
xml_doc = File.readlines("document.xml").to_s
arguments = { 'image_dir' => '/....' }
sheet = XSLT::Stylesheet.new( stylesheet, arguments )

# output to StdOut
sheet.apply( xml_doc )

# output to 'str'
str = ""
sheet.output = [ str ]
sheet.apply( xml_doc )
```

## Further Reading

- ▣ For a complete detail on REXML Parser, please refer to standard documentation for REXML Parser Documentation [↗](#).
- ▣ You can download XSLT4R from RAA Repository [↗](#).

# Web Services with Ruby - SOAP4R

## What is SOAP?

The Simple Object Access Protocol (SOAP), is a cross-platform and language-independent RPC protocol based on XML and, usually (but not necessarily) HTTP.

It uses XML to encode the information that makes the remote procedure call, and HTTP to transport that information across a network from clients to servers and vice versa.

SOAP has several advantages over other technologies like COM, CORBA etc: for example, its relatively cheap deployment and debugging costs, its extensibility and ease-of-use, and the existence of several implementations for different languages and platforms.

Please refer to our simple tutorial [SOAP](#) to understand it in detail.

This chapter makes you familiar with the SOAP implementation for Ruby (SOAP4R). This is a basic tutorial, so if you need a deep detail, you would need to refer other resources.

## Installing SOAP4R

SOAP4R is the SOAP implementation for Ruby developed by Hiroshi Nakamura and can be downloaded from –

**NOTE** – There may be a great chance that you already have installed this component.

Download SOAP

If you are aware of **gem** utility then you can use the following command to install SOAP4R and related packages.

```
$ gem install soap4r --include-dependencies
```

If you are working on Windows, then you need to download a zipped file from the above location and need to install it using the standard installation method by running *ruby install.rb*.

## Writing SOAP4R Servers

SOAP4R supports two different types of servers –

- CGI/FastCGI based (SOAP::RPC::CGIStub)
- Standalone (SOAP::RPC::StandaloneServer)

This chapter gives detail on writing a stand alone server. The following steps are involved in writing a SOAP server.

### Step 1 - Inherit SOAP::RPC::StandaloneServer Class

To implement your own stand alone server you need to write a new class, which will be child of *SOAP::StandaloneServer* as follows –

```
class MyServer < SOAP::RPC::StandaloneServer
  .....
end
```



**NOTE** – If you want to write a FastCGI based server then you need to take *SOAP::RPC::CGIStub* as parent class, rest of the procedure will remain the same.

## Step 2 - Define Handler Methods

Second step is to write your Web Services methods, which you would like to expose to the outside world.

They can be written as simple Ruby methods. For example, let's write two methods to add two numbers and divide two numbers –

```
class MyServer < SOAP::RPC::StandaloneServer
  .....

  # Handler methods
  def add(a, b)
    return a + b
  end
  def div(a, b)
    return a / b
  end
end
```

## Step 3 - Expose Handler Methods

Next step is to add our defined methods to our server. The *initialize* method is used to expose service methods with one of the two following methods –

```
class MyServer < SOAP::RPC::StandaloneServer
  def initialize(*args)
    add_method(receiver, methodName, *paramArg)
  end
end
```

Here is the description of the parameters –

Sr.No.	Parameter & Description
1	<b>receiver</b> The object that contains the methodName method. You define the service methods in the same class as the methodDef method, this parameter is <i>self</i> .
2	<b>methodName</b> The name of the method that is called due to an RPC request.
3	<b>paramArg</b> Specifies, when given, the parameter names and parameter modes.

To understand the usage of *inout* or *out* parameters, consider the following service method that takes two parameters (inParam and inoutParam), returns one normal return value (retVal) and two further parameters: *inoutParam* and *outParam* –

```
def aMeth(inParam, inoutParam)
  retVal = inParam + inoutParam
  outParam = inParam . inoutParam
  inoutParam = inParam * inoutParam
  return retVal, inoutParam, outParam
end
```

Now, we can expose this method as follows –

```
add_method(self, 'aMeth', [
  %w(in inParam),
  %w(inout inoutParam),
  %w(out outParam),
  %w(retval return)
])
```

## Step 4 - Start the Server

The final step is to start your server by instantiating one instance of the derived class and calling **start** method.

```
myServer = MyServer.new('ServerName', 'urn:ruby:ServiceName', hostname, port)
myServer.start
```

Here is the description of required parameters –

Sr.No.	Parameter & Description
1	<b>ServerName</b> A server name, you can give what you like most.
2	<b>urn:ruby:ServiceName</b> Here <i>urn:ruby</i> is constant but you can give a unique ServiceName name for this server.
3	<b>hostname</b> Specifies the hostname on which this server will listen.
4	<b>port</b> An available port number to be used for the web service.

## Example

Now, using the above steps, let us write one standalone server –

```
require "soap/rpc/standaloneserver"

begin
  class MyServer < SOAP::RPC::StandaloneServer

    # Expose our services
    def initialize(*args)
      add_method(self, 'add', 'a', 'b')
    end
  end
end
```

```

        add_method(self, 'div', 'a', 'b')
    end

    # Handler methods
    def add(a, b)
        return a + b
    end
    def div(a, b)
        return a / b
    end
end

server = MyServer.new("MyServer",
    'urn:ruby:calculation', 'localhost', 8080)
trap('INT'){
    server.shutdown
}
server.start
rescue => err
    puts err.message
end

```

When executed, this server application starts a standalone SOAP server on *localhost* and listens for requests on *port* 8080. It exposes one service methods, *add* and *div*, which takes two parameters and return the result.

Now, you can run this server in background as follows –

```
$ ruby MyServer.rb&
```

## Writing SOAP4R Clients

The *SOAP::RPC::Driver* class provides support for writing SOAP client applications. This chapter describes this class and demonstrate its usage on the basis of an application.

Following is the bare minimum information you would need to call a SOAP service –

- ▣ The URL of the SOAP service (SOAP Endpoint URL).
- ▣ The namespace of the service methods (Method Namespace URI).
- ▣ The names of the service methods and their parameters.

Now, we will write a SOAP client which would call service methods defined in above example, named *add* and *div*.

Here are the main steps to create a SOAP client.

### Step 1 - Create a SOAP Driver Instance

We create an instance of *SOAP::RPC::Driver* by calling its new method as follows –

```
SOAP::RPC::Driver.new(endPoint, nameSpace, soapAction)
```

Here is the description of required parameters –

Sr.No.	Parameter & Description
1	<b>endPoint</b> URL of the SOAP server to connect with.

2	<b>nameSpace</b> The namespace to use for all RPCs done with this SOAP::RPC::Driver object.
3	<b>soapAction</b> A value for the SOAPAction field of the HTTP header. If nil this defaults to the empty string "".

## Step 2 - Add Service Methods

To add a SOAP service method to a *SOAP::RPC::Driver* we can call the following method using *SOAP::RPC::Driver* instance –

```
driver.add_method(name, *paramArg)
```

Here is the description of the parameters –

Sr.No.	Parameter & Description
1	<b>name</b> The name of the remote web service method.
2	<b>paramArg</b> Specifies the names of the remote procedures' parameters.

## Step 3 - Invoke SOAP service

The final step is to invoice SOAP service using *SOAP::RPC::Driver* instance as follows –

```
result = driver.serviceMethod(paramArg...)
```

Here *serviceMethod* is the actual web service method and *paramArg...* is the list parameters required to pass in the service method.

### Example

Based on the above steps, we will write a SOAP client as follows –

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

require 'soap/rpc/driver'

NAMESPACE = 'urn:ruby:calculation'
URL = 'http://localhost:8080/'

begin
  driver = SOAP::RPC::Driver.new(URL, NAMESPACE)

  # Add remote service methods
  driver.add_method('add', 'a', 'b')

  # Call remote service methods
  puts driver.add(20, 30)
```

```
rescue => err
  puts err.message
end
```

## Further Readings

I have explained you just very basic concepts of Web Services with Ruby. If you want to drill down it further, then there is following link to find more details on Web Services with Ruby [↗](#).

# Ruby - Tk Guide

## Introduction

The standard graphical user interface (GUI) for Ruby is Tk. Tk started out as the GUI for the Tcl scripting language developed by John Ousterhout.

Tk has the unique distinction of being the only cross-platform GUI. Tk runs on Windows, Mac, and Linux and provides a native look-and-feel on each operating system.

The basic component of a Tk-based application is called a widget. A component is also sometimes called a window, since, in Tk, "window" and "widget" are often used interchangeably.

Tk applications follow a widget hierarchy where any number of widgets may be placed within another widget, and those widgets within another widget, ad infinitum. The main widget in a Tk program is referred to as the root widget and can be created by making a new instance of the TkRoot class.

- Most Tk-based applications follow the same cycle: create the widgets, place them in the interface, and finally, bind the events associated with each widget to a method.
- There are three geometry managers; *place*, *grid* and *pack* that are responsible for controlling the size and location of each of the widgets in the interface.

## Installation

The Ruby Tk bindings are distributed with Ruby but Tk is a separate installation. Windows users can download a single click Tk installation from ActiveState's ActiveTcl [↗](#).

Mac and Linux users may not need to install it because there is a great chance that its already installed along with OS but if not, you can download prebuilt packages or get the source from the Tcl Developer Xchange [↗](#).

## Simple Tk Application

A typical structure for Ruby/Tk programs is to create the main or **root** window (an instance of TkRoot), add widgets to it to build up the user interface, and then start the main event loop by calling **Tk.mainloop**.

The traditional *Hello, World!* example for Ruby/Tk looks something like this –

```
require 'tk'

root = TkRoot.new { title "Hello, World!" }
TkLabel.new(root) do
  text 'Hello, World!'
  pack { padx 15 ; pady 15; side 'left' }
```

```
end
Tk.mainloop
```







Here, after loading the tk extension module, we create a root-level frame using *TkRoot.new*. We then make a *TkLabel* widget as a child of the root frame, setting several options for the label. Finally, we pack the root frame and enter the main GUI event loop.



If you would run this script, it would produce the following result –



## Ruby/Tk Widget Classes

There is a list of various Ruby/Tk classes, which can be used to create a desired GUI using Ruby/Tk.


- [TkFrame](#)  Creates and manipulates frame widgets.
- [TkButton](#)  Creates and manipulates button widgets.
- [TkLabel](#)  Creates and manipulates label widgets.
- [TkEntry](#)  Creates and manipulates entry widgets.
- [TkCheckButton](#)  Creates and manipulates checkbutton widgets.
- [TkRadioButton](#)  Creates and manipulates radiobutton widgets.
- [TkListbox](#)  Creates and manipulates listbox widgets.
- [TkComboBox](#)  Creates and manipulates listbox widgets.
- [TkMenu](#)  Creates and manipulates menu widgets.
- [TkMenubutton](#)  Creates and manipulates menubutton widgets.
- [Tk.messageBox](#)  Creates and manipulates a message dialog.
- [TkScrollbar](#)  Creates and manipulates scrollbar widgets.
- [TkCanvas](#)  Creates and manipulates canvas widgets.
- [TkScale](#)  Creates and manipulates scale widgets.
- [TkText](#)  Creates and manipulates text widgets.
- [TkToplevel](#)  Creates and manipulates toplevel widgets.
- [TkSpinbox](#)  Creates and manipulates Spinbox widgets.
- [TkProgressBar](#)  Creates and manipulates Progress Bar widgets.
- [Dialog Box](#)  Creates and manipulates Dialog Box widgets.
- [Tk::Tile::Notebook](#)  Display several windows in limited space with notebook metaphor.
- [Tk::Tile::Paned](#)  Displays a number of subwindows, stacked either vertically or horizontally.

- `Tk::Tile::Separator`  Displays a horizontal or vertical separator bar.
- `Ruby/Tk Font, Colors and Images`  Understanding Ruby/Tk Fonts, Colors and Images

## Standard Configuration Options

All widgets have a number of different configuration options, which generally control how they are displayed or how they behave. The options that are available depend upon the widget class of course.

Here is a list of all the standard configuration options, which could be applicable to any Ruby/Tk widget.

There are other widget specific options also, which would be explained along with widgets. 




## Ruby/Tk Geometry Management

Geometry Management deals with positioning different widgets as per requirement. Geometry management in Tk relies on the concept of master and slave widgets.

A master is a widget, typically a top-level window or a frame, which will contain other widgets, which are called slaves. You can think of a geometry manager as taking control of the master widget, and deciding what will be displayed within.

The geometry manager will ask each slave widget for its natural size, or how large it would ideally like to be displayed. It then takes that information and combines it with any parameters provided by the program when it asks the geometry manager to manage that particular slave widget.

There are three geometry managers *place*, *grid* and *pack* that are responsible for controlling the size and location of each of the widgets in the interface.

- `grid`  Geometry manager that arranges widgets in a grid.
- `pack`  Geometry manager that packs around edges of cavity.
- `place`  Geometry manager for fixed or rubber-sheet placement.

## Ruby/Tk Event Handling

Ruby/Tk supports *event loop*, which receives events from the operating system. These are things like button presses, keystrokes, mouse movement, window resizing, and so on.

Ruby/Tk takes care of managing this event loop for you. It will figure out what widget the event applies to (did the user click on this button? if a key was pressed, which textbox had the focus?), and dispatch it accordingly. Individual widgets know how to respond to events, so for example a button might change color when the mouse moves over it, and revert back when the mouse leaves.

At a higher level, Ruby/Tk invokes callbacks in your program to indicate that something significant happened to a widget. For either case, you can provide a code block or a *Ruby Proc* object that specifies how the application responds to the event or callback.

Let's take a look at how to use the `bind` method to associate basic window system events with the Ruby procedures that handle them. The simplest form of `bind` takes as its inputs a string indicating the event name and a code block that Tk uses to handle the event.

For example, to catch the *ButtonRelease* event for the first mouse button on some widget, you'd write –

```
somewidget.bind('ButtonRelease-1') {  
  ....code block to handle this event...  
}
```

An event name can include additional modifiers and details. A modifier is a string like *Shift*, *Control* or *Alt*, indicating that one of the modifier keys was pressed.

So, for example, to catch the event that's generated when the user holds down the *Ctrl* key and clicks the right mouse button.

```
somewidget.bind('Control-ButtonPress-3', proc { puts "Ouch!" })
```

Many Ruby/Tk widgets can trigger *callbacks* when the user activates them, and you can use the *command* callback to specify that a certain code block or procedure is invoked when that happens. As seen earlier, you can specify the command callback procedure when you create the widget –

```
helpButton = TkButton.new(buttonFrame) {  
  text "Help"  
  command proc { showHelp }  
}
```

Or you can assign it later, using the widget's *command* method –

```
helpButton.command proc { showHelp }
```

Since the command method accepts either procedures or code blocks, you could also write the previous code example as –

```
helpButton = TkButton.new(buttonFrame) {  
  text "Help"  
  command { showHelp }  
}
```

You can use the following basic event types in your Ruby/Tk application –

## The configure Method

The *configure* method can be used to set and retrieve any widget configuration values. For example, to change the width of a button you can call configure method any time as follows –

```
require "tk"  
  
button = TkButton.new {  
  text 'Hello World!'  
  pack  
}  
button.configure('activebackground', 'blue')  
Tk.mainloop
```

To get the value for a current widget, just supply it without a value as follows –

```
color = button.configure('activebackground')
```



You can also call `configure` without any options at all, which will give you a listing of all options and their values.

## The `cget` Method

For simply retrieving the value of an option, `configure` returns more information than you generally want. The `cget` method returns just the current value.

```
color = button.cget('activebackground')
```

## Ruby - LDAP Tutorial

Ruby/LDAP is an extension library for Ruby. It provides the interface to some LDAP libraries like OpenLDAP, UMich LDAP, Netscape SDK, ActiveDirectory.

The common API for application development is described in RFC1823 and is supported by Ruby/LDAP.

## Ruby/LDAP Installation

You can download and install a complete Ruby/LDAP package from [SOURCEFORGE.NET](http://sourceforge.net) .

Before installing Ruby/LDAP, make sure you have the following components –

- Ruby 1.8.x (at least 1.8.2 if you want to use `ldap/control`).
- OpenLDAP, Netscape SDK, Windows 2003 or Windows XP.

Now, you can use standard Ruby Installation method. Before starting, if you'd like to see the available options for `extconf.rb`, run it with `'--help'` option.

```
$ ruby extconf.rb [--with-openldap1|--with-openldap2| \
                  --with-netscape|--with-wldap32]

$ make
$ make install
```

**NOTE** – If you're building the software on Windows, you may need to use *nmake* instead of *make*.

## Establish LDAP Connection

This is a two-step process –

### Step 1 – Create Connection Object

Following is the syntax to create a connection to a LDAP directory.

```
LDAP::Conn.new(host = 'localhost', port = LDAP_PORT)
```

- **host** – This is the host ID running LDAP directory. We will take it as *localhost*.
- **port** – This is the port being used for LDAP service. Standard LDAP ports are 636 and 389. Make sure which port is being used at your server otherwise you can use `LDAP::LDAP_PORT`.

This call returns a new `LDAP::Conn` connection to the server, *host*, on port *port*.

### Step 2 – Binding

This is where we usually specify the username and password we will use for the rest of the session.

Following is the syntax to bind an LDAP connection, using the DN, **dn**, the credential, **pwd**, and the bind method, **method** –

```
conn.bind(dn = nil, password = nil, method = LDAP::LDAP_AUTH_SIMPLE)do
  ....
end
```

You can use the same method without a code block. In this case, you would need to unbind the connection explicitly as follows –

```
conn.bind(dn = nil, password = nil, method = LDAP::LDAP_AUTH_SIMPLE)
....
conn.unbind
```

If a code block is given, *self* is yielded to the block.

We can now perform search, add, modify or delete operations inside the block of the bind method (between bind and unbind), provided we have the proper permissions.

### Example

Assuming we are working on a local server, let's put things together with appropriate host, domain, user id and password, etc.

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

require 'ldap'

$HOST = 'localhost'
$PORT = LDAP::LDAP_PORT
$SSLPORT = LDAP::LDAPS_PORT

conn = LDAP::Conn.new($HOST, $PORT)
conn.bind('cn = root, dc = localhost, dc = localdomain', 'secret')
....
conn.unbind
```

## Adding an LDAP Entry

Adding an LDAP entry is a two step process –

### Step 1 – Creating *LDAP::Mod* object

We need *LDAP::Mod* object pass to *conn.add* method to create an entry. Here is a simple syntax to create *LDAP::Mod* object –

```
Mod.new(mod_type, attr, vals)
```

- **mod\_type** – One or more option LDAP\_MOD\_ADD, LDAP\_MOD\_REPLACE or LDAP\_MOD\_DELETE.
- **attr** – should be the name of the attribute on which to operate.
- **vals** – is an array of values pertaining to *attr*. If *vals* contains binary data, *mod\_type* should be logically OR'ed (|) with LDAP\_MOD\_BVALUES.

This call returns *LDAP::Mod* object, which can be passed to methods in the *LDAP::Conn* class, such as *Conn#add*, *Conn#add\_ext*, *Conn#modify* and *Conn#modify\_ext*.

## Step 2 – Calling *conn.add* Method

Once we are ready with *LDAP::Mod* object, we can call *conn.add* method to create an entry. Here is a syntax to call this method –

```
conn.add(dn, attrs)
```

This method adds an entry with the DN, *dn*, and the attributes, *attrs*. Here, *attrs* should be either an array of *LDAP::Mod* objects or a hash of attribute/value array pairs.

### Example

Here is a complete example, which will create two directory entries –

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

require 'ldap'

$HOST = 'localhost'
$PORT = LDAP::LDAP_PORT
$SSLPORT = LDAP::LDAPS_PORT

conn = LDAP::Conn.new($HOST, $PORT)
conn.bind('cn = root, dc = localhost, dc = localdomain', 'secret')

conn.perror("bind")
entry1 = [
  LDAP.mod(LDAP::LDAP_MOD_ADD, 'objectclass', ['top', 'domain']),
  LDAP.mod(LDAP::LDAP_MOD_ADD, 'o', ['TTSKY.NET']),
  LDAP.mod(LDAP::LDAP_MOD_ADD, 'dc', ['localhost']),
]

entry2 = [
  LDAP.mod(LDAP::LDAP_MOD_ADD, 'objectclass', ['top', 'person']),
  LDAP.mod(LDAP::LDAP_MOD_ADD, 'cn', ['Zara Ali']),
  LDAP.mod(LDAP::LDAP_MOD_ADD | LDAP::LDAP_MOD_BVALUES, 'sn',
    ['ttate', 'ALI', "zero\000zero"]),
]

begin
  conn.add("dc = localhost, dc = localdomain", entry1)
  conn.add("cn = Zara Ali, dc = localhost, dc = localdomain", entry2)
rescue LDAP::ResultError
  conn.perror("add")
  exit
end
conn.perror("add")
conn.unbind
```

## Modifying an LDAP Entry

Modifying an entry is similar to adding one. Just call the *modify* method instead of *add* with the attributes to modify. Here is a simple syntax of *modify* method.

```
conn.modify(dn, mods)
```

This method modifies an entry with the DN, *dn*, and the attributes, *mods*. Here, *mods* should be either an array of *LDAP::Mod* objects or a hash of attribute/value array pairs.

## Example

To modify the surname of the entry, which we added in the previous section, we would write –

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

require 'ldap'

$HOST = 'localhost'
$PORT = LDAP::LDAP_PORT
$SSLPORT = LDAP::LDAPS_PORT

conn = LDAP::Conn.new($HOST, $PORT)
conn.bind('cn = root, dc = localhost, dc = localdomain', 'secret')

conn.perror("bind")
entry1 = [
  LDAP.mod(LDAP::LDAP_MOD_REPLACE, 'sn', ['Mohtashim']),
]

begin
  conn.modify("cn = Zara Ali, dc = localhost, dc = localdomain", entry1)
rescue LDAP::ResultError
  conn.perror("modify")
  exit
end
conn.perror("modify")
conn.unbind
```

## Deleting an LDAP Entry

To delete an entry, call the *delete* method with the distinguished name as parameter. Here is a simple syntax of *delete* method.

```
conn.delete(dn)
```

This method deletes an entry with the DN, *dn*.

## Example

To delete *Zara Mohtashim* entry, which we added in the previous section, we would write –

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

require 'ldap'

$HOST = 'localhost'
$PORT = LDAP::LDAP_PORT
$SSLPORT = LDAP::LDAPS_PORT

conn = LDAP::Conn.new($HOST, $PORT)
conn.bind('cn = root, dc = localhost, dc = localdomain', 'secret')

conn.perror("bind")
begin
  conn.delete("cn = Zara-Mohtashim, dc = localhost, dc = localdomain")
rescue LDAP::ResultError
  conn.perror("delete")
  exit
end
```

```
end
conn.perror("delete")
conn.unbind
```

## Modifying the Distinguished Name

It's not possible to modify the distinguished name of an entry with the *modify* method. Instead, use the *modrdn* method. Here is simple syntax of *modrdn* method –

```
conn.modrdn(dn, new_rdn, delete_old_rdn)
```

This method modifies the RDN of the entry with DN, *dn*, giving it the new RDN, *new\_rdn*. If *delete\_old\_rdn* is *true*, the old RDN value will be deleted from the entry.

### Example

Suppose we have the following entry –

```
dn: cn = Zara Ali,dc = localhost,dc = localdomain
cn: Zara Ali
sn: Ali
objectclass: person
```

Then, we can modify its distinguished name with the following code –

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

require 'ldap'

$HOST = 'localhost'
$PORT = LDAP::LDAP_PORT
$SSLPORT = LDAP::LDAPS_PORT

conn = LDAP::Conn.new($HOST, $PORT)
conn.bind('cn = root, dc = localhost, dc = localdomain', 'secret')

conn.perror("bind")
begin
  conn.modrdn("cn = Zara Ali, dc = localhost, dc = localdomain", "cn = Zara Mohtashim", true)
rescue LDAP::ResultError
  conn.perror("modrdn")
  exit
end
conn.perror("modrdn")
conn.unbind
```

## Performing a Search

To perform a search on a LDAP directory, use the *search* method with one of the three different search modes –

- ▣ **LDAP\_SCOPE\_BASEM** – Search only the base node.
- ▣ **LDAP\_SCOPE\_ONELEVEL** – Search all children of the base node.
- ▣ **LDAP\_SCOPE\_SUBTREE** – Search the whole subtree including the base node.

### Example

Here, we are going to search the whole subtree of entry *dc = localhost, dc = localdomain* for *person* objects –

```

#!/usr/bin/ruby -w

require 'ldap'

$HOST = 'localhost'
$PORT = LDAP::LDAP_PORT
$SSLPORT = LDAP::LDAPS_PORT

base = 'dc = localhost,dc = localdomain'
scope = LDAP::LDAP_SCOPE_SUBTREE
filter = '(objectclass = person)'
attrs = ['sn', 'cn']

conn = LDAP::Conn.new($HOST, $PORT)
conn.bind('cn = root, dc = localhost, dc = localdomain', 'secret')

conn.perror("bind")
begin
  conn.search(base, scope, filter, attrs) { |entry|
    # print distinguished name
    p entry.dn
    # print all attribute names
    p entry.attrs
    # print values of attribute 'sn'
    p entry.vals('sn')
    # print entry as Hash
    p entry.to_hash
  }
rescue LDAP::ResultError
  conn.perror("search")
  exit
end
conn.perror("search")
conn.unbind

```

This invokes the given code block for each matching entry where the LDAP entry is represented by an instance of the `LDAP::Entry` class. With the last parameter of `search`, you can specify the attributes in which you are interested, omitting all others. If you pass `nil` here, all attributes are returned same as "SELECT \*" in relational databases.

The `dn` method (alias for `get_dn`) of the `LDAP::Entry` class returns the distinguished name of the entry, and with the `to_hash` method, you can get a hash representation of its attributes (including the distinguished name). To get a list of an entry's attributes, use the `attrs` method (alias for `get_attributes`). Also, to get the list of one specific attribute's values, use the `vals` method (alias for `get_values`).

## Handling Errors

Ruby/LDAP defines two different exception classes –

- ▣ In case of an error, the `new`, `bind` or `unbind` methods raise an `LDAP::Error` exception.
- ▣ In case of `add`, `modify`, `delete` or searching an LDAP directory raise an `LDAP::ResultError`.

## Further Reading

For complete details on LDAP methods, please refer to the standard documentation for [LDAP Documentation](#).

# Ruby - Multithreading

Traditional programs have a single *thread of execution* the statements or instructions that comprise the program are executed sequentially until the program terminates.

A multithreaded program has more than one thread of execution. Within each thread, statements are executed sequentially, but the threads themselves may be executed in parallel on a multicore CPU, for example. Often on a single CPU machine, multiple threads are not actually executed in parallel, but parallelism is simulated by interleaving the execution of the threads.

Ruby makes it easy to write multi-threaded programs with the *Thread* class. Ruby threads are a lightweight and efficient way to achieve concurrency in your code.

## Creating Ruby Threads

To start a new thread, just associate a block with a call to *Thread.new*. A new thread will be created to execute the code in the block, and the original thread will return from *Thread.new* immediately and resume execution with the next statement –

```
# Thread #1 is running here
Thread.new {
  # Thread #2 runs this code
}
# Thread #1 runs this code
```

### Example

Here is an example, which shows how we can use multi-threaded Ruby program.

```
#!/usr/bin/ruby

def func1
  i = 0
  while i<=2
    puts "func1 at: #{Time.now}"
    sleep(2)
    i = i+1
  end
end

def func2
  j = 0
  while j<=2
    puts "func2 at: #{Time.now}"
    sleep(1)
    j = j+1
  end
end

puts "Started At #{Time.now}"
t1 = Thread.new{func1()}
t2 = Thread.new{func2()}
t1.join
t2.join
puts "End at #{Time.now}"
```

This will produce following result –

```
Started At Wed May 14 08:21:54 -0700 2008
func1 at: Wed May 14 08:21:54 -0700 2008
func2 at: Wed May 14 08:21:54 -0700 2008
func2 at: Wed May 14 08:21:55 -0700 2008
func1 at: Wed May 14 08:21:56 -0700 2008
func2 at: Wed May 14 08:21:56 -0700 2008
func1 at: Wed May 14 08:21:58 -0700 2008
End at Wed May 14 08:22:00 -0700 2008
```

## Thread Lifecycle

A new threads are created with *Thread.new*. You can also use the synonyms *Thread.start* and *Thread.fork*.

There is no need to start a thread after creating it, it begins running automatically when CPU resources become available.

The *Thread* class defines a number of methods to query and manipulate the thread while it is running. A thread runs the code in the block associated with the call to *Thread.new* and then it stops running.

The value of the last expression in that block is the value of the thread, and can be obtained by calling the *value* method of the *Thread* object. If the thread has run to completion, then the value returns the thread's value right away. Otherwise, the *value* method blocks and does not return until the thread has completed.

The class method *Thread.current* returns the *Thread* object that represents the current thread. This allows threads to manipulate themselves. The class method *Thread.main* returns the *Thread* object that represents the main thread. This is the initial thread of execution that began when the Ruby program was started.

You can wait for a particular thread to finish by calling that thread's *Thread.join* method. The calling thread will block until the given thread is finished.

## Threads and Exceptions

If an exception is raised in the main thread, and is not handled anywhere, the Ruby interpreter prints a message and exits. In threads, other than the main thread, unhandled exceptions cause the thread to stop running.

If a thread *t* exits because of an unhandled exception, and another thread *s* calls *t.join* or *t.value*, then the exception that occurred in *t* is raised in the thread *s*.

If *Thread.abort\_on\_exception* is *false*, the default condition, an unhandled exception simply kills the current thread and all the rest continue to run.

If you would like any unhandled exception in any thread to cause the interpreter to exit, set the class method *Thread.abort\_on\_exception* to *true*.

```
t = Thread.new { ... }
t.abort_on_exception = true
```

## Thread Variables

A thread can normally access any variables that are in scope when the thread is created. Variables local to the block of a thread are local to the thread, and are not shared.



Thread class features a special facility that allows thread-local variables to be created and accessed by name. You simply treat the thread object as if it were a Hash, writing to elements using `[]=` and reading them back using `[]`.

In this example, each thread records the current value of the variable `count` in a threadlocal variable with the key `mycount`.

[Live Demo](#)

```
#!/usr/bin/ruby

count = 0
arr = []

10.times do |i|
  arr[i] = Thread.new {
    sleep(rand(0)/10.0)
    Thread.current["mycount"] = count
    count += 1
  }
end

arr.each {|t| t.join; print t["mycount"], ", " }
puts "count = #{count}"
```

This produces the following result –

```
8, 0, 3, 7, 2, 1, 6, 5, 4, 9, count = 10
```

The main thread waits for the subthreads to finish and then prints out the value of `count` captured by each.

## Thread Priorities

The first factor that affects the thread scheduling is the thread priority: high-priority threads are scheduled before low-priority threads. More precisely, a thread will only get CPU time if there are no higher-priority threads waiting to run.

You can set and query the priority of a Ruby Thread object with `priority =` and `priority`. A newly created thread starts at the same priority as the thread that created it. The main thread starts off at priority 0.

There is no way to set the priority of a thread before it starts running. A thread can, however, raise or lower its own priority as the first action it takes.

## Thread Exclusion

If two threads share access to the same data, and at least one of the threads modifies that data, you must take special care to ensure that no thread can ever see the data in an inconsistent state. This is called *thread exclusion*.

**Mutex** is a class that implements a simple semaphore lock for mutually exclusive access to some shared resource. That is, only one thread may hold the lock at a given time. Other threads may choose to wait in line for the lock to become available, or may simply choose to get an immediate error indicating that the lock is not available.

By placing all accesses to the shared data under control of a *mutex*, we ensure consistency and atomic operation. Let's try to examples, first one without mutex and second one with mutex –

## Example without Mutex

[Live Demo](#)

```
#!/usr/bin/ruby
require 'thread'

count1 = count2 = 0
difference = 0
counter = Thread.new do
  loop do
    count1 += 1
    count2 += 1
  end
end
spy = Thread.new do
  loop do
    difference += (count1 - count2).abs
  end
end
sleep 1
puts "count1 : #{count1}"
puts "count2 : #{count2}"
puts "difference : #{difference}"
```

This will produce the following result –

```
count1 : 1583766
count2 : 1583766
difference : 0
```

[Live Demo](#)

```
#!/usr/bin/ruby
require 'thread'
mutex = Mutex.new

count1 = count2 = 0
difference = 0
counter = Thread.new do
  loop do
    mutex.synchronize do
      count1 += 1
      count2 += 1
    end
  end
end
spy = Thread.new do
  loop do
    mutex.synchronize do
      difference += (count1 - count2).abs
    end
  end
end
sleep 1
mutex.lock
puts "count1 : #{count1}"
puts "count2 : #{count2}"
puts "difference : #{difference}"
```

This will produce the following result –

```
count1 : 696591
count2 : 696591
difference : 0
```

# Handling Deadlock

When we start using *Mutex* objects for thread exclusion we must be careful to avoid *deadlock*. Deadlock is the condition that occurs when all threads are waiting to acquire a resource held by another thread. Because all threads are blocked, they cannot release the locks they hold. And because they cannot release the locks, no other thread can acquire those locks.

This is where *condition variables* come into picture. A *condition variable* is simply a semaphore that is associated with a resource and is used within the protection of a particular *mutex*. When you need a resource that's unavailable, you wait on a condition variable. That action releases the lock on the corresponding *mutex*. When some other thread signals that the resource is available, the original thread comes off the wait and simultaneously regains the lock on the critical region.

## Example

[Live Demo](#)

```
#!/usr/bin/ruby
require 'thread'
mutex = Mutex.new

cv = ConditionVariable.new
a = Thread.new {
  mutex.synchronize {
    puts "A: I have critical section, but will wait for cv"
    cv.wait(mutex)
    puts "A: I have critical section again! I rule!"
  }
}

puts "(Later, back at the ranch...)"

b = Thread.new {
  mutex.synchronize {
    puts "B: Now I am critical, but am done with cv"
    cv.signal
    puts "B: I am still critical, finishing up"
  }
}
a.join
b.join
```

This will produce the following result –

```
A: I have critical section, but will wait for cv
(Later, back at the ranch...)
B: Now I am critical, but am done with cv
B: I am still critical, finishing up
A: I have critical section again! I rule!
```

## Thread States

There are five possible return values corresponding to the five possible states as shown in the following table. The *status* method returns the state of the thread.

Thread state	Return value
Runnable	run

Sleeping	Sleeping
Aborting	aborting
Terminated normally	false
Terminated with exception	nil

## Thread Class Methods

Following methods are provided by *Thread* class and they are applicable to all the threads available in the program. These methods will be called as using *Thread* class name as follows –

```
Thread.abort_on_exception = true
```

Here is the complete list of all the class methods available –

## Thread Instance Methods

These methods are applicable to an instance of a thread. These methods will be called as using an instance of a *Thread* as follows –

```
#!/usr/bin/ruby

thr = Thread.new do # Calling a class method new
  puts "In second thread"
  raise "Raise exception"
end
thr.join # Calling an instance method join
```

Here is the complete list of all the instance methods available –

## Ruby - Built-in Functions

Since the *Kernel* module is included by *Object* class, its methods are available everywhere in the Ruby program. They can be called without a receiver (functional form). Therefore, they are often called functions.

A complete list of Built-in Functions is given here for your reference –

## Functions for Numbers

Here is a list of Built-in Functions related to number. They should be used as follows –

```
#!/usr/bin/ruby

num = 12.40
puts num.floor      # 12
puts num + 10       # 22.40
puts num.integer?   # false as num is a float.
```

[Live Demo](#)

This will produce the following result –

```
12
22.4
false
```

Assuming, **n** is a number –

## Functions for Float

Here is a list of Ruby Built-in functions especially for float numbers. Assuming we have a float number **f** –

## Functions for Math

Here is a list of Ruby Built-in math functions –

## Conversion Field Specifier

The function `sprintf( fmt[, arg...])` and `format( fmt[, arg...])` returns a string in which `arg` is formatted according to `fmt`. Formatting specifications are essentially the same as those for `sprintf` in the C programming language. Conversion specifiers (`%` followed by conversion field specifier) in `fmt` are replaced by formatted string of corresponding argument.

The following conversion specifiers are supported by Ruby's `format` –

Following is the usage example –

[🔗 Live Demo](#)

```
#!/usr/bin/ruby

str = sprintf("%s\n", "abc")   # => "abc\n" (simplest form)
puts str

str = sprintf("d=%d", 42)      # => "d=42" (decimal output)
puts str

str = sprintf("%04x", 255)     # => "00ff" (width 4, zero padded)
puts str

str = sprintf("%8s", "hello")  # => " hello" (space padded)
puts str

str = sprintf("%.2s", "hello") # => "he" (trimmed by precision)
puts str
```

This will produce the following result –

```
abc
d = 42
00ff
  hello
he
```

## Test Function Arguments

The function `test( test, f1[, f2])` performs one of the following file tests specified by the character `test`. In order to improve readability, you should use File class methods (for example, `File::readable?`) rather than this function.

Here are the file tests with one argument –

File tests with two arguments are as follows –

Following is the usage example. Assuming `main.rb` exist with read, write and not execute permissions –

[Live Demo](#)

```
#!/usr/bin/ruby

puts test(?r, "main.rb" ) # => true
puts test(?w, "main.rb" ) # => true
puts test(?x, "main.rb" ) # => false
```

This will produce the following result –

```
true
false
false
```

## Ruby - Predefined Variables

Ruby's predefined variables affect the behavior of the entire program, so their use in libraries is not recommended.

The values in most predefined variables can be accessed by alternative means.

Following table lists all the Ruby's predefined variables.

Sr.No.	Variable Name & Description
1	<b>\$!</b> The last exception object raised. The exception object can also be accessed using <code>=&gt;</code> in <i>rescue</i> clause.
2	<b>\$@</b> The stack <i>backtrace</i> for the last exception raised. The stack <i>backtrace</i> information can retrieved by <code>Exception#backtrace</code> method of the last exception.
3	<b>\$/</b> The input record separator (newline by default). <i>gets</i> , <i>readline</i> , etc., take their input record separator as optional argument.
4	<b>\$\</b>

	The output record separator (nil by default).
5	<b>\$,</b> The output separator between the arguments to <code>print</code> and <code>Array#join</code> (nil by default). You can specify separator explicitly to <code>Array#join</code> .
6	<b>\$;</b> The default separator for <code>split</code> (nil by default). You can specify separator explicitly for <code>String#split</code> .
7	<b>\$.</b> The number of the last line read from the current input file. Equivalent to <code>ARGF.lineno</code> .
8	<b>\$&lt;</b> Synonym for <code>ARGF</code> .
9	<b>\$&gt;</b> Synonym for <code>\$defout</code> .
10	<b>\$0</b> The name of the current Ruby program being executed.
11	<b>\$\$</b> The process pid of the current Ruby program being executed.
12	<b>\$?</b> The exit status of the last process terminated.
13	<b>\$:</b> Synonym for <code>\$LOAD_PATH</code> .
14	<b>\$DEBUG</b> True if the <code>-d</code> or <code>--debug</code> command-line option is specified.
15	<b>\$defout</b> The destination output for <i>print</i> and <i>printf</i> ( <code>\$stdout</code> by default).
16	<b>\$F</b>

	The variable that receives the output from <i>split</i> when -a is specified. This variable is set if the -a command-line option is specified along with the -p or -n option.
17	<b>\$FILENAME</b> The name of the file currently being read from ARGF. Equivalent to ARGF.filename.
18	<b>\$LOAD_PATH</b> An array holding the directories to be searched when loading files with the load and require methods.
19	<b>\$SAFE</b> The security level 0 → No checks are performed on externally supplied (tainted) data. (default) 1 → Potentially dangerous operations using tainted data are forbidden. 2 → Potentially dangerous operations on processes and files are forbidden. 3 → All newly created objects are considered tainted. 4 → Modification of global data is forbidden.
20	<b>\$stdin</b> Standard input (STDIN by default).
21	<b>\$stdout</b> Standard output (STDOUT by default).
22	<b>\$stderr</b> Standard error (STDERR by default).
23	<b>\$VERBOSE</b> True if the -v, -w, or --verbose command-line option is specified.
24	<b>\$- x</b> The value of interpreter option -x (x=0, a, d, F, i, K, l, p, v). These options are listed below
25	<b>\$-0</b> The value of interpreter option -x and alias of \$/.
26	<b>\$-a</b> The value of interpreter option -x and true if option -a is set. Read-only.



27	<b>\$-d</b> The value of interpreter option -x and alias of \$DEBUG
28	<b>\$-F</b> The value of interpreter option -x and alias of \$:
29	<b>\$-i</b> The value of interpreter option -x and in in-place-edit mode, holds the extension, otherwise nil. Can enable or disable in-place-edit mode.
30	<b>\$-I</b> The value of interpreter option -x and alias of \$:
31	<b>\$-I</b> The value of interpreter option -x and true if option -lis set. Read-only.
32	<b>\$-p</b> The value of interpreter option -x and true if option -pis set. Read-only.
33	<b>\$_</b> The local variable, last string read by gets or readline in the current scope.
34	<b>\$~</b> The local variable, <i>MatchData</i> relating to the last match. <code>Regex#match</code> method returns the last match information.
35	<b>\$ n (\$1, \$2, \$3...)</b> The string matched in the nth group of the last pattern match. Equivalent to <code>m[n]</code> , where <code>m</code> is a <i>MatchData</i> object.
36	<b>\$&amp;</b> The string matched in the last pattern match. Equivalent to <code>m[0]</code> , where <code>m</code> is a <i>MatchData</i> object.
37	<b>\$`</b> The string preceding the match in the last pattern match. Equivalent to <code>m.pre_match</code> , where <code>m</code> is a <i>MatchData</i> object.
38	<b>\$'</b>

	The string following the match in the last pattern match. Equivalent to <code>m.post_match</code> , where <code>m</code> is a <code>MatchData</code> object.
39	<b>\$+</b> The string corresponding to the last successfully matched group in the last pattern match.

## Ruby - Predefined Constants

The following table lists all the Ruby's Predefined Constants –

**NOTE** – `TRUE`, `FALSE`, and `NIL` are backward-compatible. It's preferable to use `true`, `false`, and `nil`.

Sr.No.	Constant Name & Description
1	<b>TRUE</b> Synonym for <code>true</code> .
2	<b>FALSE</b> Synonym for <code>false</code> .
3	<b>NIL</b> Synonym for <code>nil</code> .
4	<b>ARGF</b> An object providing access to virtual concatenation of files passed as command-line arguments or standard input if there are no command-line arguments. A synonym for <code>\$&lt;</code> .
5	<b>ARGV</b> An array containing the command-line arguments passed to the program. A synonym for <code>\$*</code> .
6	<b>DATA</b> An input stream for reading the lines of code following the <code>__END__</code> directive. Not defined if <code>__END__</code> isn't present in code.
7	<b>ENV</b> A hash-like object containing the program's environment variables. <code>ENV</code> can be handled as a hash.
8	<b>RUBY_PLATFORM</b> A string indicating the platform of the Ruby interpreter.
9	

	<b>RUBY_RELEASE_DATE</b> A string indicating the release date of the Ruby interpreter
10	<b>RUBY_VERSION</b> A string indicating the version of the Ruby interpreter.
11	<b>STDERR</b> Standard error output stream. Default value of <i>\$stderr</i> .
12	<b>STDIN</b> Standard input stream. Default value of <i>\$stdin</i> .
13	<b>STDOUT</b> Standard output stream. Default value of <i>\$stdout</i> .
14	<b>TOPLEVEL_BINDING</b> A binding object at Ruby's top level.

## Ruby - Associated Tools

### Standard Ruby Tools

The standard Ruby distribution contains useful tools along with the interpreter and standard libraries –

These tools help you debug and improve your Ruby programs without spending much effort. This tutorial will give you a very good start with these tools.

#### ▣ **RubyGems** – [↗](#)

RubyGems is a package utility for Ruby, which installs Ruby software packages and keeps them up-to-date.

#### ▣ **Ruby Debugger** – [↗](#)

To help deal with bugs, the standard distribution of Ruby includes a debugger. This is very similar to *gdb* utility, which can be used to debug complex programs.

#### ▣ **Interactive Ruby (irb)** – [↗](#)

*irb* (Interactive Ruby) was developed by Keiju Ishitsuka. It allows you to enter commands at the prompt and have the interpreter respond as if you were executing a program. *irb* is useful to experiment with or to explore Ruby.

#### ▣ **Ruby Profiler** – [↗](#)

Ruby profiler helps you to improve the performance of a slow program by finding the bottleneck.

## Additional Ruby Tools

There are other useful tools that don't come bundled with the Ruby standard distribution. However, you do need to install them yourself.

### ■ eRuby: Embedded Ruby – [↗](#)

eRuby stands for embedded Ruby. It's a tool that embeds fragments of Ruby code in other files, such as HTML files similar to ASP, JSP and PHP.

### ■ ri: Ruby Interactive Reference – [↗](#)

When you have a question about the behavior of a certain method, you can invoke ri to read the brief explanation of the method.

For more information on Ruby tool and resources, have a look at [Ruby Useful Resources](#) [↗](#).

[⏪ Previous Page](#)

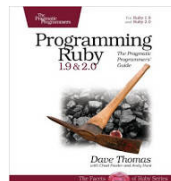
[Next Page ⏩](#)

Earn with  
G Suite



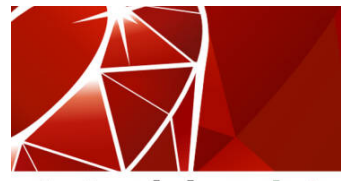
**G Suite Referral  
Program**

Ad G Suite by Google Cloud



**Ruby Useful Resources Ruby Tutorial in PDF**

tutorialspoint.com



tutorialspoint.com



**Ruby on Rails Quick  
Guide**

tutorialspoint.com



**Ruby/TK Fonts, Colors  
and Images**

tutorialspoint.com



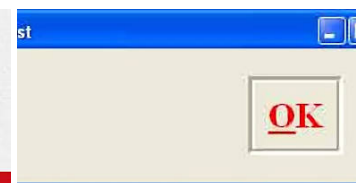
**Ruby TK Guide**

tutorialspoint.com



**Ruby on Rails  
Introduction**

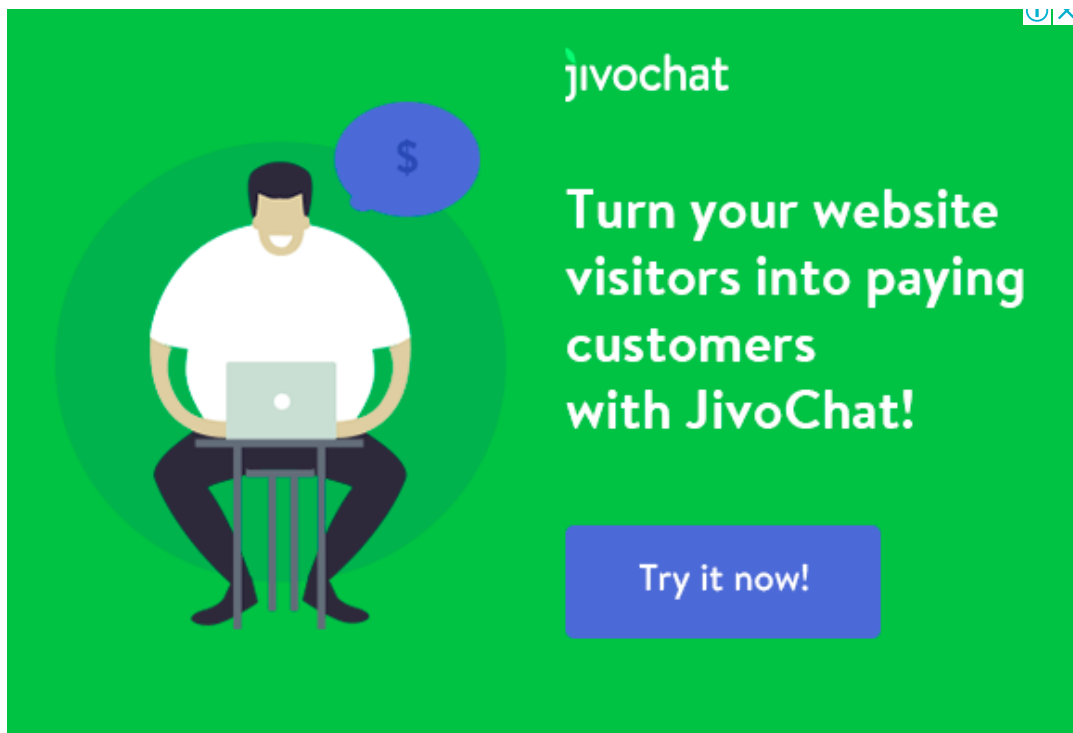
tutorialspoint.com



**Ruby/TK Button Widget**

tutorialspoint.com

Advertisements

An advertisement for JivoChat. It features a green background. On the left, there is an illustration of a man with dark hair, wearing a white t-shirt and dark pants, sitting on a stool and using a laptop. Above his head is a blue speech bubble containing a white dollar sign (\$). To the right of the illustration, the word "jivochat" is written in white lowercase letters. Below it, the text "Turn your website visitors into paying customers with JivoChat!" is written in white. At the bottom right, there is a blue button with the text "Try it now!" in white. In the top right corner of the green area, there are small icons for social media or sharing.

jivochat

Turn your website visitors into paying customers with JivoChat!

Try it now!



Tutorials Point (India) Pvt. Ltd.



YouTube

127K

**Buy E-Books Online**   
GET ACCESS TO OUR HIGH QUALITY PDFS



[FAQ's](#) | [Cookies Policy](#) | [Contact](#)

© Copyright 2018. All Rights Reserved.

Enter email for newsletter  go