

# 1 Ruby

## 1.1 Introduction

Ruby is an interpreted scripting language for object-oriented programming. Interpretive implies that a ruby application is run without first compiling the application. Variables in Ruby do not have a type; a Ruby variable may contain data of any type. Variables in Ruby may be used without any variable declarations. Ruby being an object oriented language has features such as classes, inheritance and methods. Everything in Ruby is an object including methods, strings, floats and integers. A ruby script is stored in a file with the `.rb` extension and run with the `ruby` command. First, we need to install Ruby.

## 1.2 Installing Ruby

In this section we shall install Ruby, and RubyGems. RubyGems is the standard Ruby package manager used with Ruby applications and libraries. To install Ruby, and RubyGems the procedure is as follows. Download the Ruby Windows Installer<sup>1</sup> application. Double-click on *ruby184-19.exe* application. Ruby Setup Wizard gets started. Click on Next.

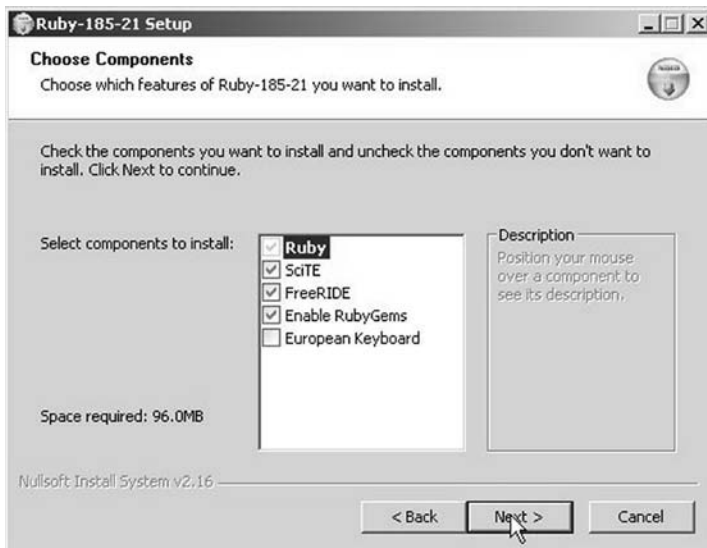
---

<sup>1</sup> Ruby Window Installer- [http://rubyforge.org/frs/?group\\_id=167](http://rubyforge.org/frs/?group_id=167)



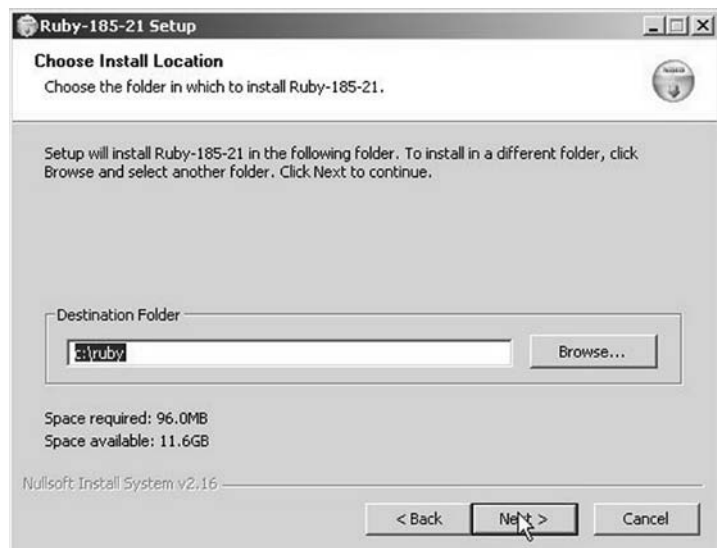
**Fig. 1.1** Ruby Setup Wizard

Accept the license agreement and click on Next. Select the default components to install, which include the RubyGems package manager, and SciTE, a Scintilla based Text Editor, and click on Next.



**Fig. 1.2** Selecting Components to install

Specify a directory to install Ruby (default is `c:/ruby`) and click on Next.



**Fig. 1.3** Specifying Installation Folder

Specify a start folder and click on Install. Ruby and RubyGems gets installed. Click on Finish to close the Ruby Setup wizard. Ruby gets installed. Directory path `c:/ruby/bin` gets added to System environment variable PATH. System environment variable RUBYOPT with value `-rubygems` gets added.

## 1.3 Creating a Ruby Application

Next, we shall create a Ruby application. For example, create a Ruby script *helloruby.rb* with the following Ruby code.

```
puts "Hello Ruby"
```

Run the Ruby script with the following command.

```
C:/>ruby helloruby.rb
```

The output from the Ruby script is as follows.

```
Hello Ruby
```

The `puts` function is used to print text. The `puts` function adds a newline after each text string. For example, modify the Ruby script to the following script, which includes a string separator.

```
puts "Hello", "Ruby"
```

The output from the Ruby script is as follows.

```
Hello
Ruby
```

If the string separator is not specified as in the following script, the strings are concatenated.

```
puts "Hello" "Ruby"
```

The output from the script is as follows.

```
HelloRuby
```

`Print` is another command to print a string. The difference between the `print` function and the `puts` function is that the `print` function does not add a newline after a string unless a newline is specified.

Ruby also provides the Ruby interactive shell to run ruby scripts. The interactive shell may be started with `irb` as shown in Figure 1.4.



**Fig. 1.4** Ruby Interactive Shell

The example Ruby scripts in this chapter are run in `irb`. A ruby script may be run from the interactive shell as shown below.

```
irb(main):001:0> puts "Hello Ruby"
```

The output is as the same as for running a `.rb` script as shown in Figure 1.5.



**Fig. 1.5** Running a Ruby Script in `irb`

Ruby also provides the `gets` function to get a string input by a user. In the Ruby interactive shell specify `gets` and press Enter.

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

Input a string, “Hello Ruby” for example, and press Enter. The string specified gets output.



```
Command Prompt - irb
irb(main):002:0> gets
Hello Ruby
=> "Hello Ruby\\n"
irb(main):003:0> _
```

**Fig. 1.6** The `gets` Function

## 1.4 Identifiers and Comments

Identifiers are names used to identify variables, methods and classes in a Ruby script. A Ruby identifier begins with a letter [a-zA-Z] or a `_` and consists of alphanumeric characters and underscores. A class name is required to begin with an uppercase letter. An identifier may not be a reserved word. Reserved words are listed below.

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

A comment begins with a `#` and a comment is defined upto the end of the line.

```
# Example of a comment
```

Documentation may be embedded in a script with `=begin` `=end`. The following listing defines documentation.

```
=begin
Example Of
Embedded Documentation
=end
```

## 1.5 Strings

A string may be specified using single quotes or double quotes. In single quotes a single quote may be escaped using `\'` and a backslash may be escaped using `\\`. In double quotes a double quote is escaped using `\''` and a backslash is escaped using `\\`. In double quotes, other characters may also be escaped such as backspace (`\b`), carriage return (`\r`), newline (`\n`), space (`\s`) and tab (`\t`). Double quotes also has the provision to evaluate embedded expressions using interpolation with `#{}` . For example, run the following ruby script.

```
puts "#{ "Hello"+"Ruby" } "
```

The output is as follows.

```
HelloRuby
```

Variables referenced in `#{}`  are required to be pre-defined. In the previous example the `+` operator is used to concatenate strings. The `*` may be used to repeat strings as in the following script.

```
puts "Hello Ruby" *3
```

The output from the script is as follows.

```
Hello RubyHello RubyHello Ruby
```

Characters are integers in Ruby. Characters may be extracted from strings as shown in the following script.

```
hello= "Hello Ruby"  
puts hello[8]
```

The character index is 0 based. The output from the script is the ASCII code for character `'b'`, 98.

Substrings may also be extracted by specifying a start index and an end index as in the following script.

```
hello= "Hello Ruby"  
puts hello[6,10]
```

The output from the script is shown below.

```
Ruby
```

Character index offsets may be specified from the end of the string with `-ve` indices. Offsets from the end of the string are 1 based and the second parameter represents the number of characters in the substring, as in the following script.

```
hello= "Hello Ruby"
puts hello[-4,4]
```

The output from the script is as follows.

```
Ruby
```

If more number of characters are specified than available in the string, the substring is created including upto the end of the string. Strings may be compared using the == operator as in the following script.

```
puts "Hello Ruby"=="Hello Ruby"
```

The output from the script is as follows.

```
true
```

Regular expressions may be used in a string. A regular expression is specified using character patterns. Some of the character patterns are discussed in Table 1.1.

**Table 1.1** Character Patterns

Pattern	Description
[]	Specifies a range. For example, [a-c] specifies a character in the range of a-c.
\w	Specifies a letter or a digit.
\W	Specifies that neither a letter nor digit should be specified.
\s	Specifies a space character.
\S	Specifies a non-space character.
\d	Specifies a digit character.
\D	Specifies a non digit character.
\b	Specifies a backspace if in a range specification. Also specifies a word boundary if not in a range specification.
\f	Form feed
\t	Horizontal tab
\v	Vertical tab
\B	Specifies a non-word boundary.

**Table 1.1** (continued)

Pattern	Description
*	Specifies 0 or more repetitions of the preceding.
+	Specifies 1 or more repetitions of the preceding.
{m,n}	Specifies at least m and at most n repetitions of the preceding.
?	Specifies at most 1 repetition of the preceding.
	Specifies that either the preceding or the next expression may match.
()	Specifies a grouping.

The % notation may be used to create string variables. The % notation is used with delimiting characters to create a string. For example, all of the following create the string “Hello Ruby”.

```
%[Hello Ruby]
%{Hello Ruby}
%(Hello Ruby)
%!Hello Ruby!
%@Hello Ruby@
```

The % notation is useful if a string contains quotes; with the % notation quotes may not be escaped. For example the following strings are equivalent.

```
"Hello \"Ruby\""
%[Hello "Ruby"]
```

## 1.6 Arrays Hashes and Ranges

Arrays are created in Ruby by listing items in [] and separating the items with a ‘,’.

```
hello = ["Hello", "Ruby"]
```



Arrays may be concatenated. For example, create an array, *hello\_array*, by concatenating another array. Run the following script in irb.

```
hello = ["Hello", "Ruby"]
hello_array=hello+["ruby", "RUBY"]
```

The output is the following array.

```
=> ["Hello", "Ruby", "ruby", "RUBY"]
```

An array may be referenced using indices, which are 0 based. For example the following array reference produces output “RUBY”.

```
hello_array[3]
```

An array may be created from another array by specifying the start index and the number of items in the array as in the following script.

```
hello_array[0,2]
```

The output from the script in irb is as follows.

```
=> ["Hello", "Ruby"]
```

An array may be created from another array by specifying a range of indices. For example, create an array from *hello\_array*, which consists of items at indices 0, 1, and 2.

```
hello_array[0..2]
```

The output is the array shown below.

```
=> ["Hello", "Ruby", "ruby"]
```

Negative indices indicate offsets from the end of the array and are 1 based. For example, create an array from the *hello\_array* with the last two elements.

```
hello_array[-2, 2]
```

The following array gets produced.

```
=> ["ruby", "RUBY"]
```

An array may be converted to a string using the *join* function. For example, create a string by joining the members of the *hello\_array* using a ‘,’.

```
hello_array.join(",")
```

The following string gets output.

```
"Hello Ruby,ruby,RUBY"
```

A string may be converted to an array using `split()`. The following script produces the array ["Hello", "Ruby", "ruby", "RUBY"].

```
"Hello Ruby, ruby, RUBY".split(", ")
```

A Hash is an associative array consisting of key-value pairs in {} brackets.

```
hello={1=>"hello", 2=>"Ruby", 3=>"ruby", 4=>"RUBY"}
```

A hash item is accessed using a key. For example, access the value of the hash entry with key 3.

```
hello[3]
```

A hash entry may be added to a hash. For example, add a hash entry with key 5.

```
hello[5]="RUby"  
hello
```

The resulting hash has the new entry appended in the beginning of the hash.

```
{5=>"RUby", 1=>"hello", 2=>"Ruby", 3=>"ruby",  
4=>"RUBY"}
```

A hash entry may be deleted using `delete`. The following command in `irb` deletes the hash entry with key 5.

```
hello.delete 5
```

Each key may occur only once in a hash. Commonly symbols are used as hash keys. The following hash is declared without using symbols.

```
hello={"a"=>"hello", "b"=>"Ruby", "c"=>"ruby",  
"d"=>"RUBY"}
```

If symbols are used, the hash is declared as follows.

```
hello={:a=>"hello", :b=>"Ruby", :c=>"ruby",  
:d=>"RUBY"}
```

A range represents a range of values. A range is specified with a start value and an end value and with 2 or three `'.'` between. Two `'.'` specify that the end value is included in the range. Three `'.'` specify that the end value is not included in the range. For example, create a range of integers between 0 and 10, excluding 10.

```
0...10
```

As another example, define a range of characters between 'a' and 'd' including 'd'.

```
'a'..'d'
```

Ranges represent increasing sequences. The following range would create an empty sequence.

```
'd'..'a'
```

To determine if a value is within a range use the `===` method. For example, create a range between 'a' and 'd' and determine if 'c' is in the range and if 'e' is in the range.

```
r = 'a'..'d'
puts r === 'c'
puts r === 'e'
```

The output is true for 'c' and false for 'e'.

## 1.7 Variables Constants and Operators

Variables in Ruby are dynamically typed, therefore, variable declarations are not used. Ruby provides four types of variables.

1. Local Variables
2. Instance Variables
3. Global Variables
4. Class Variables

The prefix of an identifier specifies the type of the variable. Different variable types are discussed in Table 1.2.

**Table 1.2** Variable Types

Variable Type	Notation	Example
Local Variable	First character [a-z] or _	var
Instance Variable	First character @	@var
Global Variable	First character \$	\$var
Class Variables	Prefix @@	@@var

Instance and global variables have the value `nil` before being initialized. Local variables are required to be initialized before being used. Class variables are available since Ruby 1.5.3 and also required to be

initialized before being used. Ruby also provides some psuedo variables: `self`, `nil`, `true` and `false`. `Self` is a global variable and refers to the current object. `Nil` is a constant and is the value assigned to uninitialized global and instance variables. The scope of a local variable is the loop, method, class, procedure object, or module in which it is defined. If the local variable is not defined in any of these constructs, the scope is the complete script. The `defined?` operator is used to check if a variable is defined as in the following script.

```
hello="Hello Ruby"
puts defined?(hello)
```

The output from the script is “local-variable”. Local variables defined in a method are not available in another method. For example, in the following script local variable *hello* is defined in the *helloRuby* method, but not in the *hello\_Ruby* method.

```
class HelloRuby
  def helloRuby

    hello= "Hello Ruby"
    return defined?(hello)
  end

  def hello_Ruby
    return defined?(hello)
  end
end
helloRubyInstance=HelloRuby.new
helloRubyInstance.helloRuby
helloRubyInstance.hello_Ruby
```

The output is local-variable for the *helloRuby* method and `nil` for the *hello\_Ruby* method.

Instance variables are defined in the scope of an object and have the initial value `nil` if uninitialized. For example define a class and define a method in the class. Define an instance variable *@hello* in the method.

```
class HelloRuby
  def hello(name)
    @hello=name
    return "Hello" + @hello
  end
end
```

The instance variable `@hello` is only available to instances of the `HelloRuby` class. For example, create a class instance and invoke the *hello* method.

```
helloRuby=HelloRuby.new
helloRuby.hello("Ruby")
```

Create another class instance and invoke the *hello* method with a different value.

```
hello_Ruby=HelloRuby.new
hello_Ruby.hello("ruby")
```

Run the following script in `irb`.

```
irb(main):001:0>class HelloRuby
  def hello(name)
    @hello=name

    return "Hello" + @hello
  end

end
helloRuby=HelloRuby.new
helloRuby.hello("Ruby")

hello_Ruby=HelloRuby.new
hello_Ruby.hello("ruby")
```

The output is “HelloRuby” for the first class instance method invocation and “Helloruby” for the second class instance. The instance variable has the value “Ruby” for the first class instance method invocation and “ruby” for the second class instance method invocation.

Global variables are available throughout a ruby script. For example, declare a global variable *\$hello* and output its value by invoking a method. Modify the variable’s value in another method and output the variable’s value as shown in the following script.

```
$hello="Ruby"
class HelloRuby
  def hello
    $hello= "Hello" +$hello
  end

  def varValue
    return $hello
  end
end
```

```
helloRuby=HelloRuby.new
helloRuby.varValue
helloRuby.hello
helloRuby.varValue
```

The \$hello variable's value is "Ruby" before being modified by invoking the hello method and "HelloRuby" after being modified.

Class variables are associated with a class and all instances of a class have the same class variable copy. The difference between class variables and global variables is that class variables are required to be initialized before being used and do not have the default value nil. As an example, create a class variable @@hello and modify the value of the variable by invoking a method of the class. The value of the variable changes for all instances of the class. The following script returns the @@hello variable value for class instance hello\_Ruby as "HelloRuby", because another class instance has modified the variable value.

```
@@hello="Ruby"
class HelloRuby
  def hello
    @@hello= "Hello" + @@hello
  end

  def varValue
    return @@hello
  end
end
helloRuby=HelloRuby.new
helloRuby.varValue
helloRuby.hello

hello_Ruby=HelloRuby.new
hello_Ruby.varValue
```

Some pre-defined system variables are defined that consist of \$ as the first character and these may not be defined as global variables. Some of these system variables are discussed in Table 1.3.

**Table 1.3** System Variables

System Variable	Description
\$!	Specifies latest error message.
\$@	Specifies error location.
\$_	Specifies string last read by gets

**Table 1.3** (continued)

System Variable	Description
\$.	Specifies line number last read by interpreter.
\$/	Specifies input record separator.
\$\	Specifies output record separator.
\$0	Specifies name of ruby script file.
\$*	Specifies the command line arguments.
\$.	Specifies line number last read by interpreter.

A constant is an identifier with a constant value and starts with an uppercase letter. Constants may be defined within classes and modules and are accessible outside the class or module. For example, define a constant *Hello* in a class and access the constant outside the class. Run the following ruby script in irb.

```
irb(main):001:0>class HelloRuby
  Hello="Hello Ruby"
end
```

```
HelloRuby::Hello
```

The output is as follows.

```
=>"Hello Ruby"
```

Constants may be reassigned value, but a warning gets generated that the constant has already been initialized.

Ruby handles all operators by converting them to methods. The method name is the same as the operator name. The '=' operator is used for assignment in following Ruby. example.

```
var=1
strvar="String Variable"
hello=Hello.new
```

Ruby supports the +=, -=, \*=, /=, \*\*= operators. Ruby also supports multiple assignments as shown below.

```
x,y,z='Hello', 'Ruby', 'ruby'
```

```
puts x
puts y
puts z
```

The output is as follows.

```
Hello
Ruby
Ruby
```

Arrays may be created using `%w()` or `%{}`. For example, the following script outputs “Ruby”.

```
array=%w(Hello Ruby ruby)
puts array[1]
```

The `||=` operator is used for conditional assignment. If a variable value is `nil` the value specified with `||=` is assigned to the variable. For example, the following script outputs “default value”.

```
var=nil
var||="default value"
puts var
```

Ruby also provides symbols. A symbol is a variable prefixed with a colon (`:`), which is stored with a unique id, for example `:var1`. Symbols are like constants and are used for comparison in Rails as they require less processing than strings.

## 1.8 Classes

Ruby is an object oriented language and a class represents the template from which objects may be created. An object is an instance of a class. A class consists of variables and methods. A class definition starts with `class` and ends with `end`. A class name is required to begin with a capital letter. The following script defines a class *Hello*, which consists of a method *hello*.

```
class Hello

  def hello
    return "Hello Ruby"
  end
end
```



A class is instantiated with the `new` method. For example, create an instance of the class `Hello`.

```
hello=Hello.new
```

Using the class object invoke the method `hello`.

```
hello.hello
```

The output from the method invocation is “Hello Ruby”. Classes in Ruby support inheritance. For example create another class `Msg`, which extends class `Hello`. Extending a class is denoted with ‘<’. Define a method `msg` in class `Msg` that return a string. Create an instance of class `Msg` and invoke the `msg` method. As the `Msg` class extends the `Hello` class, an instance of class `Msg` is also an instance of class `Hello`. Invoke the `hello` method of class `Hello` with an instance of class `Msg`. Run the following script .

```
class Hello

  def hello
    return "Hello Ruby"
  end
end

class Msg <Hello
  def msg
    return "Hello ruby"
  end
end

msg=Msg.new
msg.msg
msg.hello
```

The output from invoking the `msg` method is “Hello ruby” and the output from invoking the `hello` method is “Hello Ruby”. Ruby does not support multiple inheritance, therefore, a class may extend only one other class.

A `initialize` function may be defined to initialize a class. The `initialize` function is invoked after a class instance is created. For example, define a class with the `initialize` function. Initialize an instance variable `@hello` in the `initialize` function. Output the value of the instance variable by invoking another method of a class instance.

```
class Hello
  @hello
```

```
def initialize(hello)
  @hello=hello
end

def hello
  return @hello
end

hello=Hello.new("Hello Ruby")
hello.hello
```

The output from the Ruby script is “Hello Ruby”.

## 1.9 Methods

Methods in Ruby begin with `def` and end with `end`. The following method takes a name parameter and returns a string.

```
class Hello

  def hello(name)
    return "Hello" +name
  end

end
```

A method is invoked with an instance of the class in which the method is defined. The `hello` method of class `Hello` may be invoked as follows.

```
helloObj=Hello.new
helloObj.hello("Deepak")
```

The output from the method invocation is “HelloDeepak”. Method names in Ruby should begin with a lowercase letter. By default, methods return the last statement in the method. Therefore, the following method, which does not have a return statement, would also return a “Hello ...” string.

```
def hello(name)
  "Hello" +name
end
```

Method parameters may be assigned default values. For example, in the following method definition parameter name is assigned a default value.

---

```
def hello(name="John")
  return "Hello" +name
end
```

If the method is invoked without an argument, the default value is used. The following script outputs “HelloRuby”.

```
class Hello

  def hello(name="Ruby")
    return "Hello" +name
  end

end

helloObj=Hello.new
helloObj.hello
```

Ruby has the provision to define methods with a variable number of arguments by preceding the last parameter of a method with an asterisk (\*). For example, define method *hello* to take a variable number of arguments. The following script outputs “Hello Ruby,ruby, RUBY”.

```
class Hello

  def hello(*name)
    return "Hello " +name.join(',')
  end

end

helloObj=Hello.new
helloObj.hello("Ruby", "ruby", "RUBY")
```

The asterisk operator may also precede an Array argument in a method invocation. In the following script the hello method is invoked with an array using the \* operator.

```
class Hello

  def hello(name1,name2,name3)
    return "Hello " +name1+", "+name2+", "+name3
  end

end

helloObj=Hello.new
array=["Ruby", "ruby", "RUBY"]
helloObj.hello(*array)
```

The output from the script is “Hello Ruby, ruby, RUBY”. The parentheses in method invocation may be omitted. The hello method may be invoked with arguments as follows.

```
helloObj.hello "Ruby", "ruby", "RUBY"
```

Parentheses are required if another method is to be invoked on the method invocation result. For example if a method returns an array and the order of the elements in the array is to be reversed, parentheses are required as shown below.

```
array= helloObj.hello("Ruby", "ruby", "RUBY").reverse
```

A hash may be used as an argument to a method. For example, define a method hello and invoke the method with a hash as shown below.

```
class Hello

  def hello(name)
    return "Hello " +name[:c]
  end

end

helloObj=Hello.new
helloObj.hello :a=>"Ruby", :b=>"ruby", :c=>"RUBY"
```

The output from the method invocation is “Hello RUBY”.Methods in Ruby are public, by default. The access may be restricted by `public`, `private` and `protected` methods, `Public`, `private`, and `protected` are not keywords, but methods that operate on a class.

For example, in the following class/method definition hello is declared as a private method.

```
class Hello
  def hello(name)
    return "Hello " +name
  end
  private :hello
end
```

If `private` is invoked without arguments, all methods following `private` are set to private, as in the following example.

```
class Hello

  private
  def methodA
  end

end
```

```

    def methodB
    end
end

```

Methods *methodA* and *methodB* are set to private. A method may also be set to private with the method `private_class_method`.

```
private_class_method :hello
```

Private methods may only be accessed within the class they are declared or a subclass of the class. For example, if class *Hello* defines a private method *hello*, and *helloObj1* is an instance of class *Hello*, *helloObj1* may only access non-private methods of class *Hello* even though *helloObj1* is an instance of class *Hello*. In the following script, method *hello* is private to class *Hello*, and may only be invoked within the class.

```

class Hello

  def hello(name)
    return "Hello " +name
  end
private :hello

  def helloRuby
    hello "Ruby"
  end
end

helloObj1=Hello.new
helloObj1.helloRuby
helloObj1.hello "Ruby"

```

The output from the script is the string “Hello Ruby” for the *helloRuby* method invocation, which invokes private method *hello*. When the *hello* method is invoked directly by an instance of class *Hello* an error gets output: “NoMethodError: private method ‘hello’ called...”.

Protected methods also may be accessed within the defining class and subclasses of the class. The difference between private methods and protected methods is that a protected method may be invoked with an explicit receiver while a private method may be invoked with only *self* as the receiver, which implies that a protected method may be invoked by an instance of the defining class and by an instance of a subclass of the defining class while a private method may only be invoked within the context of the defining class or a subclass of the defining class. In the preceding example, method *hello* may be invoked with an instance of class

Hello, as shown below, or an instance of a sub-class of Hello, if method hello is protected.

```
class Hello

  def hello(name)
    return "Hello " +name
  end
protected :hello
helloObj1=Hello.new

helloObj1.hello "Ruby"

end
```

Ruby provides accessor methods for instance variables. Without the accessor methods getter/setter methods would have to be used. For example, getter/setter methods are used in the following listing to access an instance variable.

```
class Catalog

  def initialize(catalogid)
    @catalogid=catalogid
  end

  def getCatalogid
    @catalogid
  end

  def setCatalogid(catalogid)
    @catalogid=catalogid
  end

end

catalog=Catalog.new("catalog1")
catalog.getCatalogid
catalog.setCatalogid("catalog2")
catalog.getCatalogid
```

The output from the Ruby script is as follows.

```
"catalog1"
"catalog2"
"catalog2"
```

The `attr_accessor` function provides the getter/setter functionality. In the following script, the `attr_accessor` method is used on the *catalogid* instance variable.

```
class Catalog

  def initialize(catalogid)
    @catalogid=catalogid
  end
  attr_accessor :catalogid

end

catalog=Catalog.new("catalog1")
catalog.catalogid
catalog.catalogid="catalog2"
catalog.catalogid
```

More than one instance variables may be specified in an `attr_accessor` function.

```
attr_accessor :var1, :var2
```

If only getter functionality is required use function `attr_reader`, and if only setter functionality is required use the `attr_writer` function.

Ruby provides Singleton methods, which are defined only for an object of a class. For example, define a class `Hello` with a method `hello`. Create an instance of the class and define a singleton method for the instance of the class.

```
class Hello

  def hello
    return "Hello Ruby"
  end

end

helloObj=Hello.new
helloObj.hello

def helloObj.hello(name)
  "Hello"+ name
end
helloObj.hello("ruby")
```

The script returns “Hello Ruby” for the invocation of the `hello` method and “Helloruby” for the invocation of the singleton method `hello(name)`, which is defined for the `helloObj` object.

## 1.10 Procs and Blocks

Proc objects are blocks of code bound to a set of local variables. A block:

```
{ |x| ... }
```

is equivalent to:

```
do |x| .
```

A Proc object is created using the `Proc.new` method. Create a proc that outputs a Hello message.

```
hello=Proc.new{|name| puts "Hello "+name}
hello.call("Ruby")
```

The output from the script in `irb` is “Hello Ruby”.

If a local variable specified in a Proc object is previously specified, and the Proc object is invoked with a variable value, the previously specified variable value gets changed. In the following script variable `x` value gets changed to 10 after invoking the Proc object.

```
x=1
proc = Proc.new {|x| puts x }
proc.call(10)
puts x
```

The parameters of a Proc object are specified in the `||` in the beginning of the block. The code following the parameters is run when the Proc is invoked. A Proc is invoked with the `call` method, which takes the arguments to the Proc object and returns the last expression evaluated in the block. More than one parameters may be specified in a Proc object. The following script, which invokes a Proc object with 3 parameters, outputs the message “Hello Ruby, ruby, RUBY”.

```
hello=Proc.new{|name1, name2, name3| puts "Hello
"+name1+", "+name2+", "+name3}
hello.call("Ruby", "ruby", "RUBY")
```



The parameters may be omitted from a Proc object as in the following script, which outputs “Hello Ruby”.

```
hello=Proc.new{ puts "Hello Ruby"}
hello.call()
```

A method may be invoked with a Proc object argument. For example, create a class Hello and a method helloMthd, which takes 2 parameters. Create a Proc object, create an instance of the class and invoke the method with the Proc object as shown in following listing.

```
class Hello
  def helloMthd(param1, param2)
    return param1.call(param2)
  end

end

helloProc=Proc.new{|name| puts "Hello "+name}
helloObj=Hello.new
helloObj.helloMthd(helloProc, "Ruby")
```

The output from invoking the helloMthd method with a Proc object is “Hello Ruby”. If a Proc.new object in a method contains a return statement, invoking the Proc object returns from the enclosing method. In the following script, a method creates a Proc object with Proc.new. In the Proc object a return statement is specified. The Proc object is invoked in the method. When the method is invoked, the Proc object gets invoked, and the method invocation returns.

```
class Hello
  def hello()
    helloProc=Proc.new{return "Return from Proc"}
    helloProc.call()
    puts "Hello Ruby"
  end

end

helloObj=Hello.new
helloObj.hello()
```

The output from the script is “Return from Proc”. The “Hello Ruby” string is not output. The Kernel module provides a method called `proc` or

`lambda`, which is equivalent to `Proc.new`, but which does not return from the enclosing method. If the preceding script is run with the `proc` method, instead of `Proc.new`, the output is “Hello Ruby”. Another difference between `Proc.new` and the `proc` method is that the `proc` method checks for the number of arguments, while `Proc.new` doesn’t. For example, a `Proc.new` block, which defines 2 parameters, may be invoked with 3 arguments as in the following script.

```
hello=Proc.new{|name1, name2| puts "Hello "+name1}
hello.call("Ruby", "ruby", "RUBY")
```

The output is “Hello Ruby”.

In contrast, if the `proc` method is used to create a `Proc` object and the `Proc` object is invoked with a different number of arguments than specified, an error gets generated. For example, the following script creates a `Proc` object with the `proc` method that defines 2 parameters, and when the `Proc` object is invoked with 3 arguments an error gets generated: “ArgumentError: wrong number of arguments (3 for 2)”.

```
hello=proc{|name1, name2| puts "Hello "+name1}
hello.call("Ruby", "ruby", "RUBY")
```

The `Proc.new` method may be used without a block, if invoked in a method and the method has an attached block, as in the following script.

```
def hello
  Proc.new
end
helloProc = hello { "hello ruby" }
helloProc.call
```

A block of code may be used with a method without using `Proc.new` to create a `Proc` object. When a block is appended to a method call, Ruby converts the block of code to a `Proc` object without a name. The `Proc` object may be invoked in the method using the `yield` method, which is equivalent to an explicit call to an explicit `Proc` object. In the following listing method `hello` is invoked with a block. Ruby converts the block to a `Proc` object, which may be called using the `yield` method.

```
def hello
  yield
```

```
    yield
  end

hello {puts "Hello Ruby"}
```

The output from the script is as follows.

```
Hello Ruby
Hello Ruby
```

The ampersand operator (&) may be used to explicitly convert between a block and a Proc object. If an & is prepended to the last parameter of a method and a block attached with the method, the block gets converted to a Proc object and gets assigned to the last argument. In the following example, the last argument of the hello method is prepended with an &. When the method invocation is attached with a block, the block gets converted to a Proc object and gets assigned to the last argument of the method. The call method may be invoked on the Proc object 'name' in the method definition. The yield method may still be used to invoke the Proc object.

```
def hello(msg, &name)

  name.call(msg)
  yield(msg)
end

hello ("Ruby") {|name| puts "Hello " +name}
```

The output from the Ruby script is as follows.

```
Hello Ruby
Hello Ruby
```

The argument prepended with & isn't really an argument, but meant to convert a block of code to a Proc object. A method may not be invoked with a Proc object where a block is expected. For example, if the hello method in the preceding script is invoked with a Proc object instead of a block, as in the following listing, an ArgumentError gets generated.

```
def hello(msg,&name)
  name.call(msg)
  yield(msg)
end

hello("Ruby", proc {|name| puts "Hello " +name})
```

But, a Proc object may be converted to a block and a method that expects a block invoked with the converted block. A Proc object is converted to a block by prepending the Proc object with an `&`. In the following script, the `procObj` Proc object is prepended with a `&` in the `hello` method invocation.

```
def hello(msg,&name)

  name.call(msg)
  yield(msg)
end

procObj=proc {|name| puts "Hello " +name}
hello("Ruby", &procObj)
```

The output is the same as invoking the method with a block.

## 1.11 Control Structures and Iterators

Ruby provides control structures to run code conditionally. A conditional branch evaluates a test expression and evaluates code in a block depending on whether the expression evaluates to true or false. The `if` control structure is used evaluate a block of code if the expression following `if` evaluates to true as shown in the following example.

```
var1=nil
if var1==nil
  var1="Nil Variable"
end
```

The output is "Nil Variable". The test expression and code block may be put on the same line using `then`.

```
var1=nil
if var1==nil then var1="Nil Variable" end
```

The `if` expression may also be used as follows.

```
var1=nil
var1="Nil Variable" if var1==nil
```

The `unless` expression evaluates a block of code if an expression evaluates to false.

```
var1=nil
unless var1!=nil
  "Variable is Nil"
end
```

The output is "Nil Variable". The `if-elsif-else` expression evaluates a series of expressions. For example, the following `if-elsif-else` script outputs "Var1 is nil".

```
var1=nil
if var1==1
  "Var1 is 1"
elsif var1==2
  "Var1 is 2"
elsif var1==5
  "Var1 is 5"
else
  "Var1 is nil"
end
```

The short-if statement is used to evaluate one expression if a Boolean expression is true and another expression if the Boolean expression is false.

```
var1=5
(var1==nil)? nil : "Var1 is not nil"
```

The preceding Ruby script outputs "Var1 is not nil". The `case` statement is used to test a sequence of conditions. The following script tests `name` with different strings and outputs "Ruby".

```
name="Ruby"
case name
  when "RUBY"
    puts "RUBY"
  when "ruby"
    puts "ruby"
  when "Ruby"
    puts "Ruby"
end
```

The `while` statement runs a block of code while a specified condition is true. The following script outputs an integer and increments the integer while the integer is not 10.

```
var=1
while var!=10
  puts var
  var +=1
end
```

The `until` statement is a negated while. The following script outputs an integer and increments an integer until the integer is 10.

```
var=1
until var==10
  puts var
  var +=1
end
```

Ruby provides four methods to exit a while/until loop: `break`, `next`, `redo`, and `return`. The `break` exits the loop. In the following script, integers are output only upto 7.

```
var=1
while var!=10
  if var==8
    break
  end
  puts var
  var +=1
end
```

The `next` statement invokes the next iteration of a loop. In the following script, which has a `next` statement, integers 2 to 10 are output except integer 8, because the next iteration is invoked if `var` value is 8.

```
var=1
while var!=10
  var +=1
  if var==8
    next
  end
  puts var
end
```

The `redo` statement restarts the current iteration again. The following script restarts current iteration if `var` value is 8. The output is integers 1 to 9.

```
var=1
while var!=10
  puts var
  var +=1
  if var==8
    redo
  end
end

end
```

A `return` statement in a loop exits the loop and also the method that contains the loop. The following script iterates the while loop twice.

```
class Hello

  def hello
    var=1
    while var!=10
      puts "Hello Ruby"
      var+=1
      if var==3
        return "Hello Ruby"
      end
    end
  end

end

hello=Hello.new
hello.hello
```

The `for` statement iterates over a collection without using indices. The collection may be a hash, an array, a range or any other collection. The following script iterates over an array and outputs a Hello message for each element in the collection.

```
array =["Ruby", "ruby", "RUBY"]

for name in array
  puts "Hello"+ name
end
```

The output is as follows.

```
"Hello Ruby"
"Hello ruby"
Hello RUBY"
```

A collection may also be iterated using the `each` method. The following script also produces the same output as the preceding script.

```
array = ["Ruby", "ruby", "RUBY"]

array.each do |name|
  puts "Hello " + name
end
```

A string type provides a method `each_byte`, which iterates over each character in the string. The following snippet outputs ASCII character codes for the characters in the “RUBY” string.

```
str="RUBY"

str.each_byte do |c|
  puts c
end
```

Ruby provides another iterator for string type, `each_line`, which iterates over each line in a string.

```
str="RUBY\nRuby\nruby"

str.each_line do |l|
  puts l
end
```

The output from the code snippet is as follows.

```
RUBY
Ruby
Ruby
```

The `each` method for a string type is the same as the `each_line` method. The `retry` statement restarts the iteration from the beginning. The following script, outputs “Hello Ruby” twice.

```
array = ["Ruby", "ruby", "RUBY"]
c=0
array.each do |name|
  if name=="ruby" and c==1
    retry
  end
  puts "Hello " + name
end
```



```
c +=1  
end
```

The `redo` statement is used to restart the current iteration. The following script does not output a string if `c` is 1.

```
array = ["Ruby", "ruby", "RUBY"]  
c=0  
array.each do |name|  
  if c==1  
    c +=1  
    redo  
  end  
  puts c  
  puts "Hello "+ name  
  c +=1  
end
```

Ruby provides the `n.times` do iterator for `n` iterations. For example, the following iteration outputs 0, 1, 2, 3.

```
4.times do |num|  
  puts num  
end
```

## 1.12 Exception Handling

Exceptions are conditions in the running of code that prevent the code from running. An `Exception` is an instance of class `Exception` or a sub-class of `Exception`. In the section on methods, we discussed that if a private method of a class is invoked with an instance of the class, a `NoMethodError` gets generated. `NoMethodError` is a sub-class of `NameError` class, which is a sub-class of `StandardError` class, which is a sub-class of the `Exception` class. Ruby provides exception handling mechanism with `begin/end` block. If an exception is raised in a `begin/end` block Ruby provides the `rescue` clause to handle the exception. Multiple `rescue` clauses may be specified in a `begin/end` block to handle different error conditions. An `ensure` clause may also be specified that consists of statements that are run whether an exception occurs or not. The format of a `begin/end` block is as follows.

```
begin

  rescue Exception1
    Statements to run when an exception of type
    Exception1 occurs
  rescue Exception2
    Statements to run when exception of type
    Exception2 occurs.
  ensure
    Statements to run whether an exception occurs or
    not.

end
```

A reference to the exception object associated with the latest exception is available in the global variable `$!`. In the following script, a `NoMethodError` gets generated when a private method a class is invoked. An error message is output in the rescue statement.

```
class Hello

  def hello(name)
    return "Hello " +name
  end

  private :hello

end

begin
  helloObj1=Hello.new
  helloObj1.hello "Ruby"
  rescue NoMethodError
    $stderr.print "The NoMethodError has been generated:
" + $!
  end
```

The output from the script is as follows.

The `NoMethodError` has been generated: private method  
hello called for #<Hello:>

If no exception class is specified in the rescue clause, the `StandardError` exception is the default. Multiple exception classes may be specified in a rescue class, and a local variable may be specified to receive the matched exception. For example, in the following script multiple exception classes have been assigned to a rescue clause and also a local variable has been assigned to the rescue class.

```
class Hello

  def hello(name)
    return "Hello " +name
  end
private :hello

end

begin
helloObj1=Hello.new
helloObj1.hello "Ruby"
rescue NoMethodError, SyntaxError =>error
$stderr.print error
end
```

The output from the script is as follows.

```
private method hello called for #<Hello:>
```

Parameters to the rescue clause may be expressions that return an Exception class. Exceptions may also be raised explicitly using the raise method. The raise method has one of the following syntaxes.

```
raise
raise( aString )
raise( anException [, aString [ anArray ] ] )
```

With no arguments, raise raises the exception in !\$ or raises a RuntimeError if !\$ is nil. With a single argument, raise raises a RuntimeError with the string message. With the third syntax, the first parameter is the Exception class or a sub-class of the Exception class. The optional second parameter is string message associated with the exception. The optional third parameter is an array of callback information. In the following script, an exception of type Exception is raised in the hello method and the rescue clause outputs the error message.

```
class Hello

  def hello(name)
    raise Exception, "An exception has been generated
in the hello method"
    return "Hello " +name
  end

end
```

```
begin
  helloObj1=Hello.new
  helloObj1.hello "Ruby"
rescue NoMethodError, Exception =>error
  $stderr.print error
end
```

The output from the script is as follows.

```
"An exception has been generated in the hello
method".
```

The raise method is available in the kernel module.

## 1.13 Modules

A module is a collection of classes, methods, variables, and constants. A module is defined with the following syntax.

```
module

end
```

A module is similar to a class in that it is a collection of methods, variables, and constants. But, a module is different from a class, because a module may not be instantiated or sub-classed. Members of a module are referenced with the `::` notation. For example, if class `Class1` is in module `Module1`, the class is referenced as `Module1::Class1`. Modules provide multiple inheritance with mixins. A module may be included in a class, thus, the members of the module become the members of the class. A module is included in a class with the `include` statement. If the module is another file, first import the module with a `require` statement.

```
require Module1
include Module1
```

## 1.14 Comparing Ruby with PHP

Both PHP and Ruby are interpreted scripting languages. Both PHP and Ruby are object-oriented and provide classes, methods, and class inheritance. Ruby is more object-oriented than PHP; in Ruby everything is an object. In both Ruby and PHP, a class may extend one other class; single inheritance. In both Ruby and PHP access to classes and methods

may be public, protected or private. The PHP script runs on the web server and output may be viewed in a web browser. For server-side-scripting three components are required; PHP Installation, Web Server, and a Web Browser. PHP is dynamically typed; variables are not declared, just as in Ruby. Ruby provides the constant nil corresponding to PHP type NULL. Both Ruby and PHP provide the constants TRUE and FALSE. Both Ruby and PHP support expression interpolation for double-quoted strings using `#{};` expressions enclosed in `#{} in a double quoted string` are evaluated and replaced with the result. Both Ruby and PHP support exception handling. Both Ruby and PHP may be embedded in HTML, the syntax though is different. PHP code is embedded using `<? ?>` and Ruby code is embedded using `<% %>`, or `<%= %>` to output to a browser. Ruby and PHP are different in some other aspects too. Ruby is a strongly typed language, which means that explicit conversions have to be performed between data types, unlike PHP, which performs the type conversions automatically. Strings, numbers, arrays, and hashes are objects in Ruby unlike in PHP. Integers in Ruby may contain underscores as markers, which are not evaluated by the parser. Ruby provides symbols, which PHP doesn't. In Ruby parentheses are optional in method invocation, unlike in PHP. Ruby provides control structures if, else and elsif corresponding to PHP's control structures if, else and elseif. Corresponding to PHP's while, do-while, for and foreach, Ruby provides n.times do, while, begin-end-until, for and .each do. Ruby does not support abstract classes or interfaces, which PHP does. Almost everything in Ruby gets converted to a method call.

## 1.15 Comparing Ruby with Java

Ruby is similar to Java in that both are object-oriented languages and are strongly typed. But, Ruby is dynamically typed, whereas Java is statically typed; in Ruby type declarations are not used while in Java type declarations are required. Both Java and Ruby provide inheritance and have public, private and protected methods. Ruby is simpler than Java and faster than Java too. Ruby is different from Java in a number of features. The differences between Java and Ruby are discussed in Table 1.4.

**Table 1.4** Comparing Ruby with Java

Feature	Ruby	Java
Interpreted/Compiled	Ruby is an interpreted scripting language and is run directly.	Java applications are required to be compiled before running.
Defining Blocks	Ruby defines a class/method block using the end keyword.	Java uses braces to define a class/method block.
Importing packages/modules	The require statement is used to import a class or a module.	The import statement is used to import a package or a class.
Multiple Inheritance.	Uses mixins for multiple inheritance.	Uses interfaces for multiple inheritance.
Typed Variables	Variables do not have an explicit type associated.	Variables have an explicit type.
Constructor	Constructor is the initialize method.	Constructor is the name of the class.
Class Instantiation.	A class Class1 is instantiated as follows: class1=Class1.new	A class Class1 is instantiated as follows: class1=new Class1()
Configuration file	YAML files are used.	XML files
Null value	nil	null
Casting	No casting.	Casting is used.
Type declarations.	No type declarations. Variables are dynamically typed.	Variables are statically typed.

**Table 1.4** (continued)

Feature	Ruby	Java
Objects	Everything is an object including numbers.	Objects
Parentheses in method invocation.	Parentheses in method invocation are optional.	Parentheses in method invocation.
Member variables.	All member variables are private.	Member variables.

## 1.16 Summary

In this chapter we installed Ruby. We discussed the Ruby syntax. We compared Ruby with PHP another commonly used scripting language. We also compared Ruby with Java.



<http://www.springer.com/978-3-540-73144-3>

Ruby on Rails for PHP and Java Developers

Vohra, D.

2007, XVI, 394 p. 202 illus., Softcover

ISBN: 978-3-540-73144-3