UNIT –IV: RUBY

Objective:

To be familiar with the development of Ruby scripting language.

Syllabus:

Introduction to ruby,variables,types,simple I/O, Controls, Arrays, Hashes, Methods, Classes, Iterators.

Outcomes:

Students will be able to

Understand the basics of Ruby.

Interpret ruby variables and data types.

Execute different statements and control structures of Ruby.

Understand the importance of classes,iterators in program execution.

➢ Introduction to Ruby:

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

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

Ruby is "A Programmer's Best Friend".

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

Features ofRuby

- 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.

RubyInstallationonWindows

Here are the steps to install Ruby on a Windows machine.

NOTE: You may have different versions available at the time of installation.

- Download a zipped file having latest version of Ruby. Follow **Download Link**.
- After having downloaded the Ruby archive, unpack it and change into the newly created directory:
- Double-click the Ruby1.6.7.exe file. The Ruby installation wizard starts.
- Click Next to move to the Important Information page of the wizard and keep moving till Ruby installer completes installing Ruby.

You may need to set some environment variables if your installation has not setup them appropriately.

- If you use Windows 9x, add the following lines to your c:\autoexec.bat: set PATH="D:\(ruby install directory)\bin;%PATH%"
- Windows NT/2000 users need to modify their registries.
    - Click Control Panel | System Properties | Environment Variables.
    - Under System Variables, select Path and click EDIT.
    - Add your Ruby directory to the end of the Variable Value list and click OK.
    - Under System Variables, select PATHEXT and click EDIT.
    - Add .RB and .RBW to the Variable Value list and click OK.
- After installation, make sure everything is working fine by issuing the following command on the command-line:

```
$ruby -v

ruby 1.6.7
```

- If everything is fine, this should output the version of the installed Ruby interpreter as shown above. You may have installed different version, so it will display a different version.

RubyCommandLineOptions

Ruby is generally run from the command line in the following way:

```
$ ruby [ options ] [.] [ programfile ] [ arguments ... ]
```

The interpreter can be invoked with any of the following options to control the environment and behavior of the interpreter.

Option	Description
-a	Used with -n or -p to split each line. Check -n and -p options.
-c	Checks syntax only, without executing program.
-C dir	Changes directory before executing (equivalent to -X).
-d	Enables debug mode (equivalent to -debug).
-F pat	Specifies pat as the default separator pattern ($;) used by split.
-e prog	Specifies prog as the program from the command line. Specify multiple -e options for multiline programs.
-h	Displays an overview of command-line options.
-i [ext]	Overwrites the file contents with program output. The original file is saved with the extension ext. If ext isn't specified, the original file is deleted.
-I dir	Adds dir as the directory for loading libraries.
-K [kcode]	Specifies the multibyte character set code (e or E for EUC (extended Unix code); s or S for SJIS (Shift-JIS); u or U for UTF-8; and a, A, n, or N for ASCII).
-l	Enables automatic line-end processing. Chops a newline from input lines and appends a newline to output lines.
-n	Places code within an input loop (as in while gets; ... end).
-0[octal]	Sets default record separator ($/) as an octal. Defaults to \0 if octal not specified.
-p	Places code within an input loop. Writes $_ for each iteration.
-r lib	Uses *require* to load *lib* as a library before executing.
-s	Interprets any arguments between the program name and filename arguments fitting the pattern -xxx as a switch and defines the corresponding variable.

-T [level]	Sets the level for tainting checks (1 if level not specified).
-v	Displays version and enables verbose mode
-w	Enables verbose mode. If program file not specified, reads from STDIN.
-x [dir]	Strips text before #!ruby line. Changes directory to *dir* before executing if *dir* is specified.
-X dir	Changes directory before executing (equivalent to -C).
-y	Enables parser debug mode.
--copyright	Displays copyright notice.
--debug	Enables debug mode (equivalent to -d).
--help	Displays an overview of command-line options (equivalent to -h).
--version	Displays version.
--verbose	Enables verbose mode (equivalent to -v). Sets $VERBOSE to true.
--yydebug	Enables parser debug mode (equivalent to -y).

Single character command-line options can be combined. The following two lines express the same meaning:

```
$ruby -ne 'print if /Ruby/' /usr/share/bin


$ruby -n -e 'print if /Ruby/' /usr/share/bin
```

Popular RubyEditors

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.

➢ Ruby Variables:

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

There are four types of variables in Ruby:

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

1.Local variables

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

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

2.Class variables

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

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

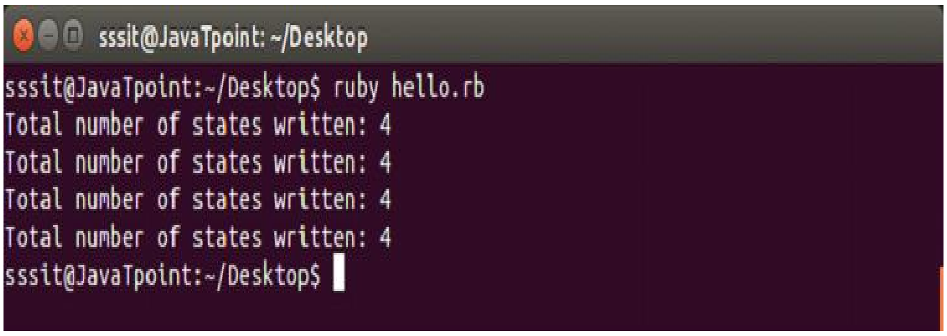
Example:

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

class States
  @@no_of_states=0
  def initialize(name)
    @states_name=name
    @@no_of_states += 1
  end
  def display()
   puts "State name #@state_name"
   end
   def total_no_of_states()
     puts "Total number of states written: #@@no_of_states"
   end
end

# Create Objects
```

```ruby
first=States.new("Assam")
second=States.new("Meghalaya")
third=States.new("Maharashtra")
fourth=States.new("Pondicherry")

# Call Methods
first.total_no_of_states()
second.total_no_of_states()
third.total_no_of_states()
fourth.total_no_of_states()
```

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

Output:



```
sssit@JavaTpoint:~/Desktop$ ruby hello.rb
Total number of states written: 4
Total number of states written: 4
Total number of states written: 4
Total number of states written: 4
sssit@JavaTpoint:~/Desktop$
```

3. Instance variables:

An instance variable name starts with a @ sign. It belongs to one instance of the class and can be accessed from any instance of the class within a method. They only have limited access to a particular instance of a class.

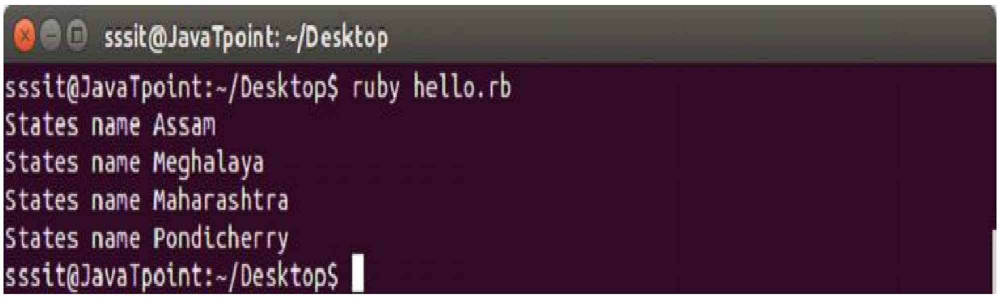
They don't need to be initialize. An uninitialized instance variable will have a nil value.

Example:

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

class States
  def initialize(name)
    @states_name=name
  end
  def display()
    puts "States name #@states_name"
   end
end

# Create Objects
first=States.new("Assam")
second=States.new("Meghalaya")
third=States.new("Maharashtra")
fourth=States.new("Pondicherry")

# Call Methods
```

```
first.display()
second.display()
third.display()
fourth.display()
```

Output:



```
sssit@JavaTpoint:~/Desktop$ ruby hello.rb
States name Assam
States name Meghalaya
States name Maharashtra
States name Pondicherry
sssit@JavaTpoint:~/Desktop$
```

Global variables

A global variable name starts with a $ sign. Its scope is globally, means it can be accessed from any where in a program.

An uninitialized global variable will have a nil value. It is advised not to use them as they make programs cryptic and complex.

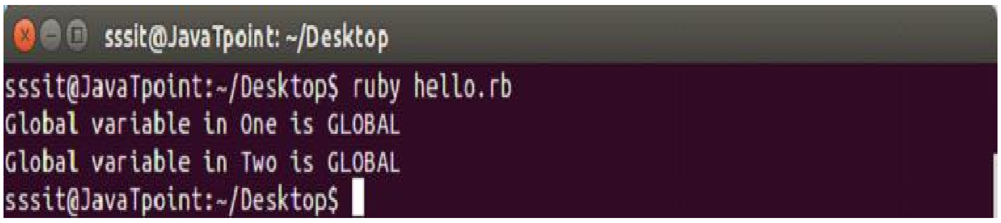
There are a number of predefined global variables in Ruby.

Example:

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

$global_var = "GLOBAL"
class One
  def display
     puts "Global variable in One is #$global_var"
  end
end
class Two
  def display
     puts "Global variable in Two is #$global_var"
  end
end

oneobj = One.new
oneobj.display
twoobj = Two.new
twoobj.display
```

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

Output:



```
sssit@JavaTpoint:~/Desktop$ ruby hello.rb
Global variable in One is GLOBAL
Global variable in Two is GLOBAL
sssit@JavaTpoint:~/Desktop$
```

Summary

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

➢ **Ruby Data types**

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

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

Numbers

Integers and floating point numbers come in the category of numbers.

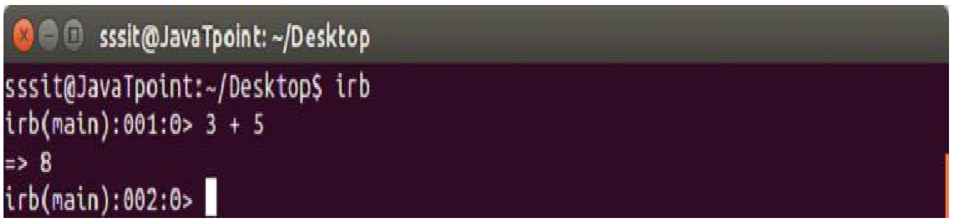
Integers are held internally in binary form. Integer numbers are numbers without a fraction. According to their size, there are two types of integers. One is Bignum and other is Fixnum.

Class	Description	Example
Fixnum	They are normal numbers	1
Bignum	They are big numbers	111111111111
Float	Decimal numbers	3.0
Complex	Imaginary numbers	4 + 3i

Rational	They are fractional numbers	9/4
BigDecimal	Precision decimal numbers	6.0

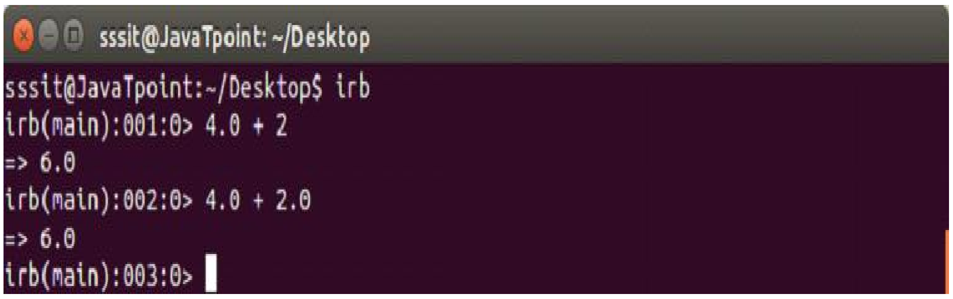
Example:

- In a calculation if integers are used, then only integers will be returned back.



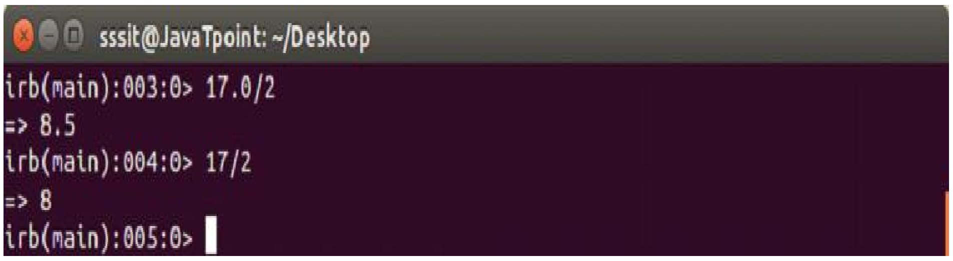
```
sssit@JavaTpoint: ~/Desktop
sssit@JavaTpoint:~/Desktop$ irb
irb(main):001:0> 3 + 5
=> 8
irb(main):002:0>
```

- In a calculation if float type is used, then only float will be returned back.



```
sssit@JavaTpoint: ~/Desktop
sssit@JavaTpoint:~/Desktop$ irb
irb(main):001:0> 4.0 + 2
=> 6.0
irb(main):002:0> 4.0 + 2.0
=> 6.0
irb(main):003:0>
```

- In case of dvision, following output will appear.

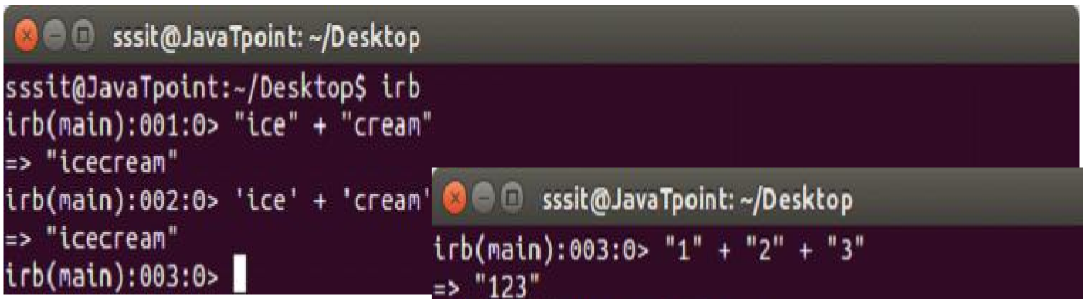


```
sssit@JavaTpoint: ~/Desktop
irb(main):003:0> 17.0/2
=> 8.5
irb(main):004:0> 17/2
=> 8
irb(main):005:0>
```

Strings

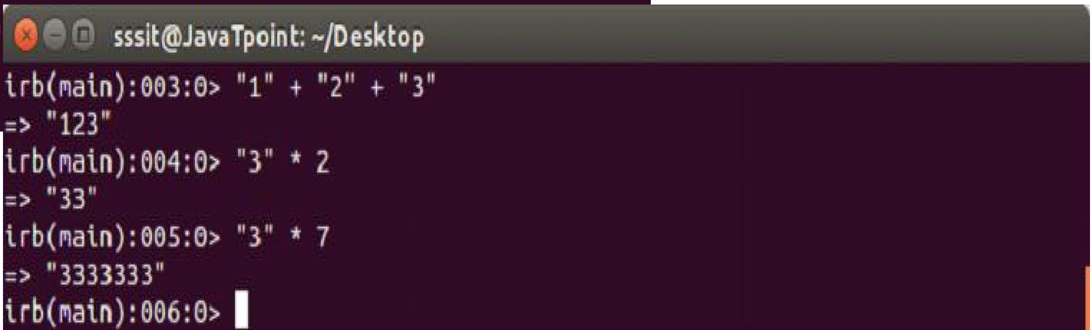
A string is a group of letters that represent a sentence or a word. Strings are defined by enclosing a text within single (') or double (") quote.

Example: Two strings can be concatenated using + sign in between them.



```
sssit@JavaTpoint: ~/Desktop
sssit@JavaTpoint:~/Desktop$ irb
irb(main):001:0> "ice" + "cream"
=> "icecream"
irb(main):002:0> 'ice' + 'cream'
=> "icecream"
irb(main):003:0>
```

-
- Multiplying a number



```
sssit@JavaTpoint: ~/Desktop
irb(main):003:0> "1" + "2" + "3"
=> "123"
irb(main):004:0> "3" * 2
=> "33"
irb(main):005:0> "3" * 7
=> "3333333"
irb(main):006:0>
```

string with a number will repeat the string as many times.

Symbols

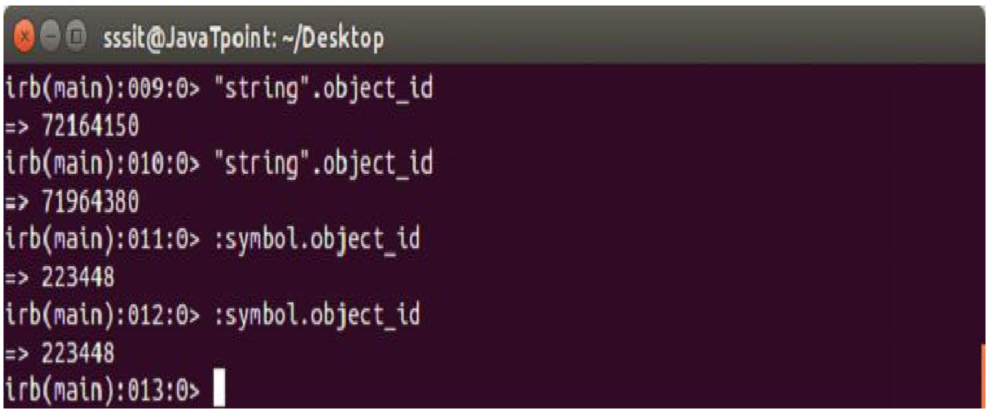
Symbols are like strings. A symbol is preceded by a colon (:). For example,

1. :abcd

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

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

Example:



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

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

Hashes

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

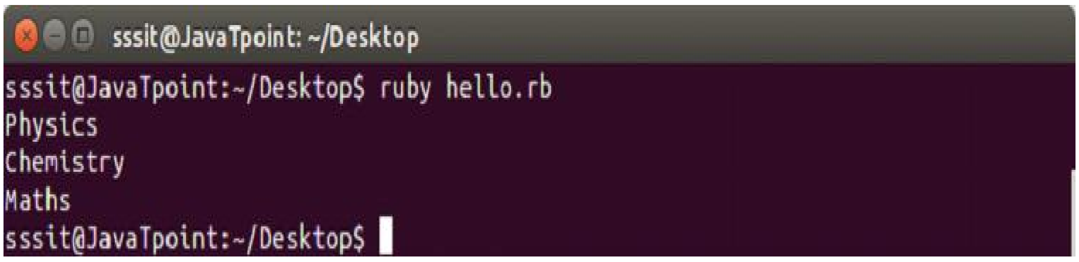
{"Akash" => "Physics", "Ankit" => "Chemistry", "Aman" => "Maths"}

Example:

```
#!/usr/bin/ruby

```

```
data = {"Akash" => "Physics", "Ankit" => "Chemistry", "Aman" => "Maths"}
puts data["Akash"]
puts data["Ankit"]
puts data["Aman"]
```

Output:



```
sssit@JavaTpoint:~/Desktop$ ruby hello.rb
Physics
Chemistry
Maths
sssit@JavaTpoint:~/Desktop$
```

Arrays

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

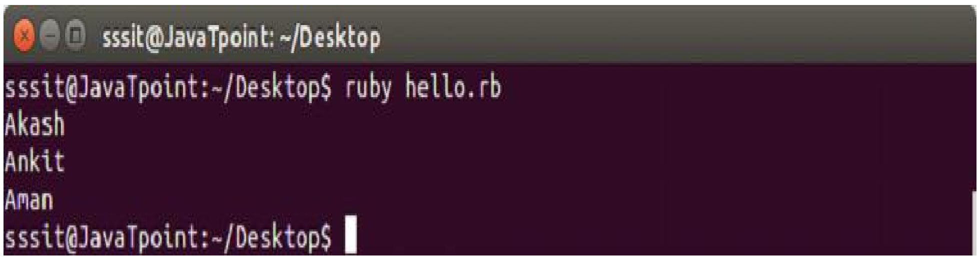
```
["Akash", "Ankit", "Aman"]
```

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

Example:

```
#!/usr/bin/ruby

data = ["Akash", "Ankit", "Aman"]
puts data[0]
puts data[1]
puts data[2]
```

Output:



```
sssit@JavaTpoint:~/Desktop$ ruby hello.rb
Akash
Ankit
Aman
sssit@JavaTpoint:~/Desktop$
```

➢ **Control Statements:**

Ruby If-else Statement

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

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

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

example

```
a = gets.chomp.to_i
if a >= 18
  puts "You are eligible to vote."
else
  puts "You are not eligible to vote."
end

```

Ruby if else if (elsif)

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

Example:

```
a = gets.chomp.to_i
if a <50
  puts "Student is fail"
elsif a >= 50 && a <= 60
  puts "Student gets D grade"
elsif a >= 70 && a <= 80
  puts "Student gets B grade"
elsif a >= 80 && a <= 90
  puts "Student gets A grade"
elsif a >= 90 && a <= 100
```

```
  puts "Student gets A+ grade"
end
```

Ruby ternary Statement

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

Syntax:

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

Example:

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

Ruby Case Statement

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

Syntax:

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

Example:

```
#!/usr/bin/ruby
print "Enter your day: "
day = gets.chomp
case day
when "Tuesday"
  puts 'Wear Red or Orange'
when "Wednesday"
  puts 'Wear Green'
when "Thursday"
  puts 'Wear Yellow'
 when "Friday"
  puts 'Wear White'
 when "Saturday"
  puts 'Wear Black'
else
  puts "Wear Any color"
end
```

Ruby for Loop

Ruby for loop iterates over a specific range of numbers. Hence, for loop is used if a program has fixed number of iterations.

Ruby for loop will execute once for each element in expression.

Syntax:

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

Ruby for loop using range

Example:

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

Ruby for loop using array

Example:

```
x = ["Blue", "Red", "Green", "Yellow", "White"]
for i in x do
  puts i
end
```

Ruby while Loop

The Ruby while loop is used to iterate a program several times. If the number of iterations is not fixed for a program, while loop is used.

Ruby while loop executes a condition while a condition is true. Once the condition becomes false, while loop stops its execution.

Syntax:

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

Example:

```
#!/usr/bin/ruby
x = gets.chomp.to_i
```

```
while x >= 0
  puts x
  x -=1
end
```

Ruby Until Loop

The Ruby until loop runs until the given condition evaluates to true. It exits the loop when condition becomes true. It is just opposite of the while loop which runs until the given condition evaluates to false.

The until loop allows you to write code which is more readable and logical.

Syntax:

```
until conditional
   code
end
```

Example:

```
i = 1
until i == 10
    print i*10, "\n"
    i += 1
end
```

➢ Ruby Methods

Ruby methods prevent us from writing the same code in a program again and again. It is a set of expression that returns a value.

Ruby methods are similar to the functions in other languages. They unite one or more repeatable statements into one single bundle.

Defining Method

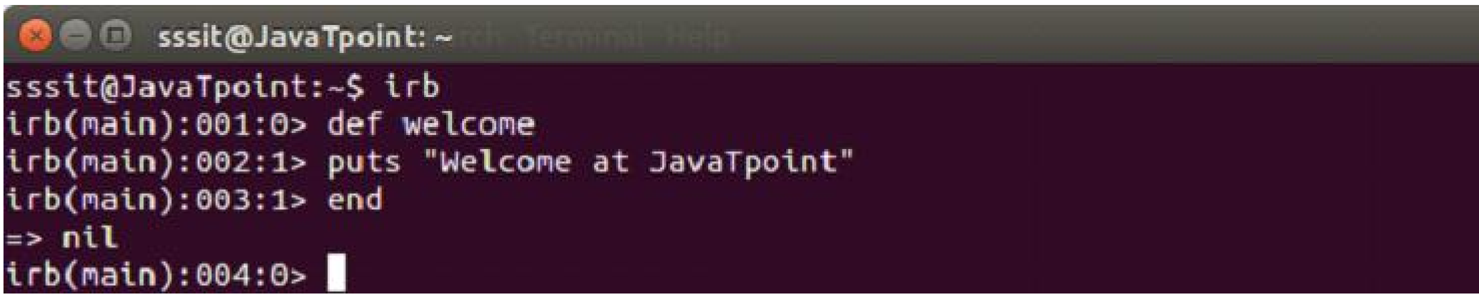
To use a method, we need to first define it. Ruby method is defined with the **def** keyword followed by method name. At the end we need to use **end** keyword to denote that method has been defined.

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

Syntax:

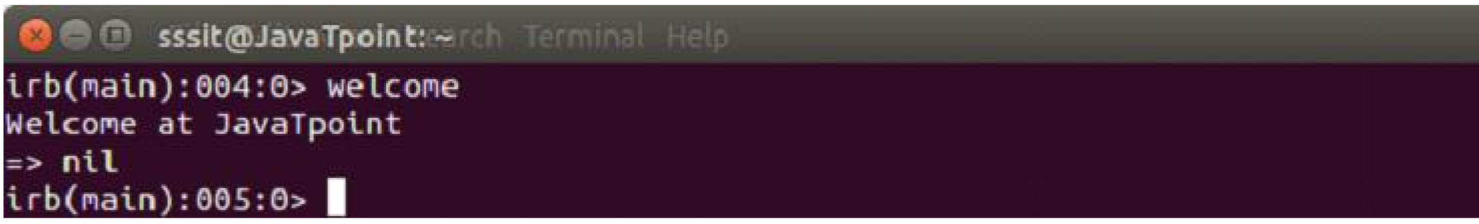
```
def methodName
    code...
end
```

Example:


```
sssit@JavaTpoint:~$ irb
irb(main):001:0> def welcome
irb(main):002:1> puts "Welcome at JavaTpoint"
irb(main):003:1> end
=> nil
irb(main):004:0> 
```

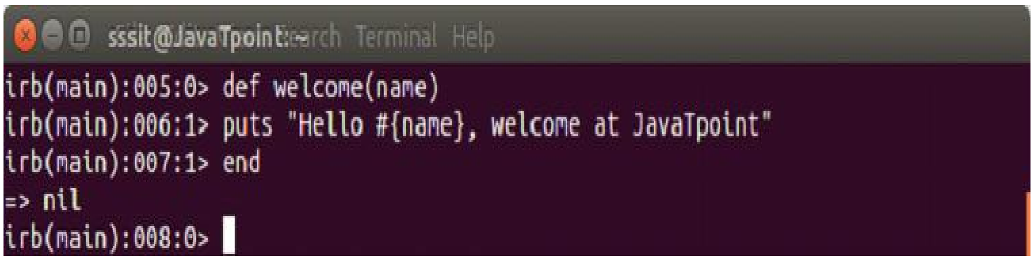
Here, we have defined a method **welcome** using def keyword. The last line end keyword says that we are done with the method defining.

Now let's call this method. A method is called by just writing its name.


```
irb(main):004:0> welcome
Welcome at JavaTpoint
=> nil
irb(main):005:0> 
```

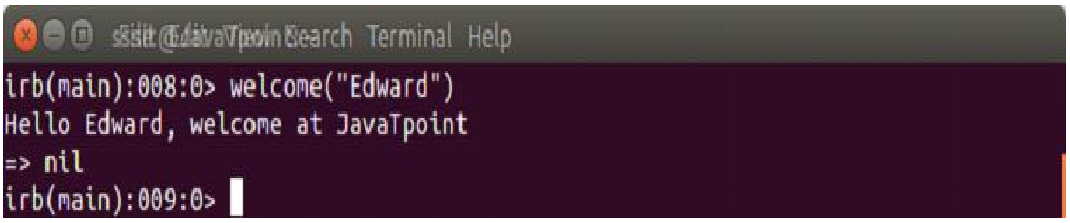
Defining Method with Parameter

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


```
irb(main):005:0> def welcome(name)
irb(main):006:1> puts "Hello #{name}, welcome at JavaTpoint"
irb(main):007:1> end
=> nil
irb(main):008:0> 
```

Here, #{name} is a way in Ruby to insert something into string. The bit inside the braces is turned into a string.

Let's call the method by passing a parameter **Edward.**


```
irb(main):008:0> welcome("Edward")
Hello Edward, welcome at JavaTpoint
=> nil
irb(main):009:0> 
```

Ruby Class and Object

Here, we will learn about Ruby objects and classes. In object-oriented programming language, we design programs using objects and classes.

Object is a physical as well as logical entity whereas class is a logical entity only.

Ruby Object

Object is the default root of all Ruby objects. Ruby objects inherit from **BasicObject** (it is the parent class of all classes in Ruby) which allows creating alternate object hierarchies.

Object mixes in the Kernel module which makes the built-in Kernel functions globally accessible.

Creating object

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

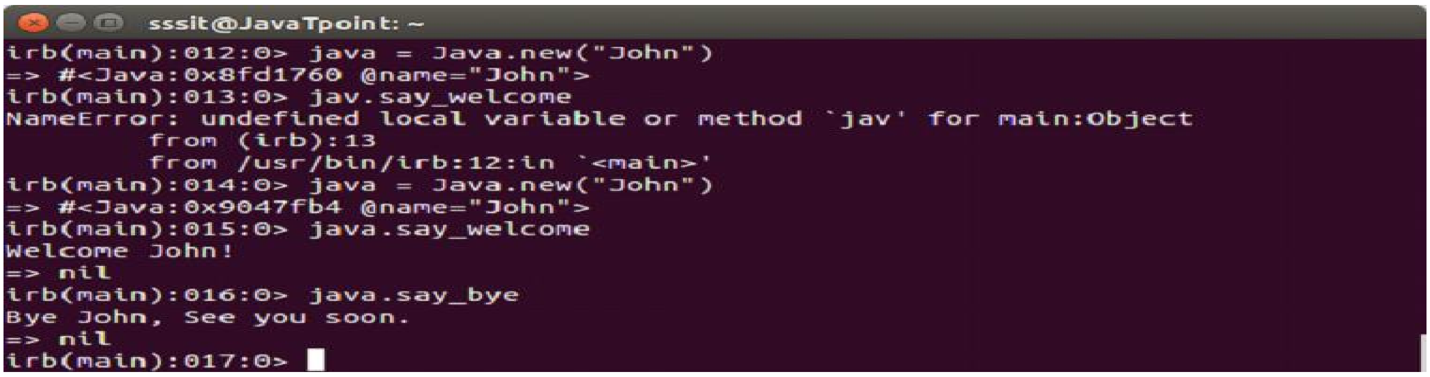
Ruby objects are instances of the class.

Syntax:

1. objectName = className.new

Example:

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


```
sssit@JavaTpoint: ~
irb(main):012:0> java = Java.new("John")
=> #<Java:0x8fd1760 @name="John">
irb(main):013:0> jav.say_welcome
NameError: undefined local variable or method `jav' for main:Object
        from (irb):13
        from /usr/bin/irb:12:in `<main>'
irb(main):014:0> java = Java.new("John")
=> #<Java:0x9047fb4 @name="John">
irb(main):015:0> java.say_welcome
Welcome John!
=> nil
irb(main):016:0> java.say_bye
Bye John, See you soon.
=> nil
irb(main):017:0>
```

java = Java.new("John")

➢ Ruby Class

Each Ruby class is an instance of class **Class**. Classes in Ruby are first-class objects.

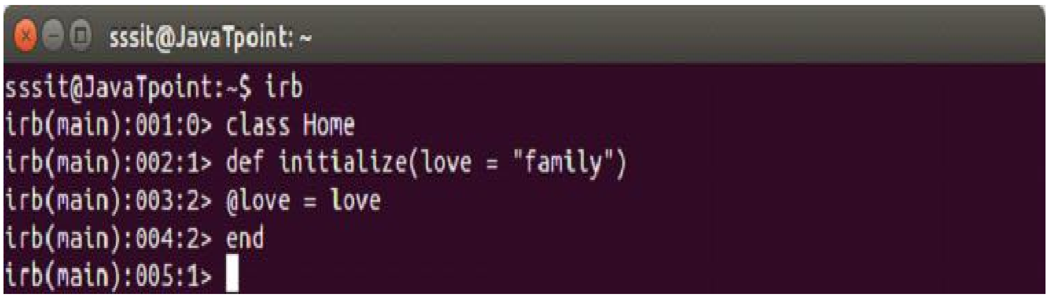
Ruby class always starts with the keyword **class** followed by the class name. Conventionally, for class name we use CamelCase. The class name should always start with a capital letter. Defining class is finished with **end** keyword.

Syntax:

1. **class** ClassName

```
    codes...
end
```

Example:



```
sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ irb
irb(main):001:0> class Home
irb(main):002:1> def initialize(love = "family")
irb(main):003:2> @love = love
irb(main):004:2> end
irb(main):005:1>
```

In the above example, we have created a class **Home** using class keyword. The **@love** is an instance variable, and is available to all methods of class Home.

➢ Ruby Iterators

Iterator is a concept used in object-oriented language. Iteration means doing one thing many times like a loop.

The loop method is the simplest iterator. They return all the elements from a collection, one after the other. Arrays and hashes come in the category of collection.

Ruby Each Iterator

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

Syntax:

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

Here collection can be any array, range or hash.

Example:

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

Ruby Times Iterator

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

Syntax:

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

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

Example:

```
#!/usr/bin/ruby
5.times do |n|
  puts n
end
```

Ruby Upto and Downto Iterators

An upto iterator iterates from number x to number y.

Syntax:

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

Example:

```
#!/usr/bin/ruby
1.upto(5) do |n|
  puts n
end
```

Ruby Step Iterator

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

Syntax:

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

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

Example:

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

Ruby Each_Line Iterator

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

Example:

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