Ruby Programming

Student Name :____ PRN__________________

Course Teacher: Mrs. Sharayu Bonde and Mrs Priyanka Gadade, Govt. College of Engg., Jalgaon

1. Overview

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

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.

1.1 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 clean and easy syntax that allows a new developer to learn 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 installed in Windows and POSIX environments.
- 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.

2. Syntax

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

```
#!/usr/bin/ruby -w
puts "Hello, Ruby!";
```

Here, 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.

2.1 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)
```

2.2 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.

2.3 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 (_).

2.4 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
```

3. Classes and Objects

Ruby is a perfect Object Oriented Programming Language. 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.

3.1 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.

3.2 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
```

3.3 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.

3.4 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")
```

3.5 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


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!
```

4. If-else, case

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

4.1 Ruby if...else Statement

Syntax

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

if expressions are used for conditional execution. The values *false* and *nil* are false, and everything else are true. Notice, Ruby uses elsif, not else if nor elif.

Executes *code* if the *conditional* is true. If the *conditional* is not true, *code* specified in the else clause is executed.

An if expression's *conditional* is separated from code by the reserved word *then*, a newline, or a semicolon.

Example

```
#!/usr/bin/ruby

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

end
x is 1
```

4.2 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
```

Example

```
#!/usr/bin/ruby


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

This will produce the following result:

```
little child
```

5. 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.

5.1 Ruby while Statement

Syntax

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

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

Example

```
#!/usr/bin/ruby

$i = 0
$num = 5

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

This will produce the following result:

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

5.2 Ruby for Statement

Syntax

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

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

Example

```
#!/usr/bin/ruby


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

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

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

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

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

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

Example

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

This will produce the following result:

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

5.3 Ruby break Statement

Syntax

```
break
```

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

Example

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

This will produce the following result:

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

5.4 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

```
#!/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
```

6. 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

```
#!/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
```

6.1 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*.

6.2 Ruby return Statement

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

Syntax

```
return [expr[`,' expr...]]
```

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

Example

```
return
OR
return 12
OR
return 1,2,3
```

Have a look at this example:

```
#!/usr/bin/ruby

def  test
        i = 100
        j = 200
        k = 300
return i, j, k
end

var = test
puts var
```

This will produce the following result:

```
100
200
300
```

6.3 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.

7. 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.

7.1 Expression Substitution

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

```
#!/usr/bin/ruby

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.
```

8. 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.

8.1 Creating Arrays

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

```
names = Array.new
```

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

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

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

```
#!/usr/bin/ruby

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

This will produce the following result:

```
20
20
```

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

```
#!/usr/bin/ruby
names = Array.new(4, "mac")
puts "#{names}"
```

This will produce the following result:

```
macmacmacmac
```

You can also use a block with new, populating each element with what the block evaluates to:

```
#!/usr/bin/ruby

nums = Array.new(10) { |e| e = e * 2 }
puts "#{nums}"
```

This will produce the following result:

```
024681012141618
```

There is another method of Array, []. It works like this:

```
nums = Array.[](1, 2, 3, 4,5)
```

One more form of array creation is as follows:

```
nums = Array[1, 2, 3, 4,5]
```

The *Kernel* module available in core Ruby has an Array method, which only accepts a single argument. Here, the method takes a range as an argument to create an array of digits:

```
#!/usr/bin/ruby

digits = Array(0..9)
puts "#{digits}"
```

9. 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.

9.1 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).

9.2 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
```

9.3 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
```

9.4 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 avriables
             @width, @height = w, h
       end
end
```

9.5 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}"
```

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

```
Area of the box is : 200
```

Name and Sign of Course Teacher