Introduction to Ruby

Outlines

Introduction to MVC

Introduction to Ruby

Introduction to MVC

- **Model–View–Controller** is an architectural pattern commonly used for developing user interfaces that divides an application into three interconnected parts.

- The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development.

- Programming languages like Java, C#, Python, Ruby, PHP have MVC frameworks that are used in web application development straight out of the box.

- Some web MVC frameworks places almost the entire model, view and controller logic on the server. This is reflected in frameworks such as Django, Rails and ASP.NET MVC. In this approach, the client sends either hyperlink requests or form submissions to the controller and then receives a complete and updated web page (or other document) from the view; the model exists entirely on the server.

- Other frameworks such as AngularJS, EmberJS, JavaScriptMVC and Backbone allow the MVC components to execute partly on the client (also see Ajax).

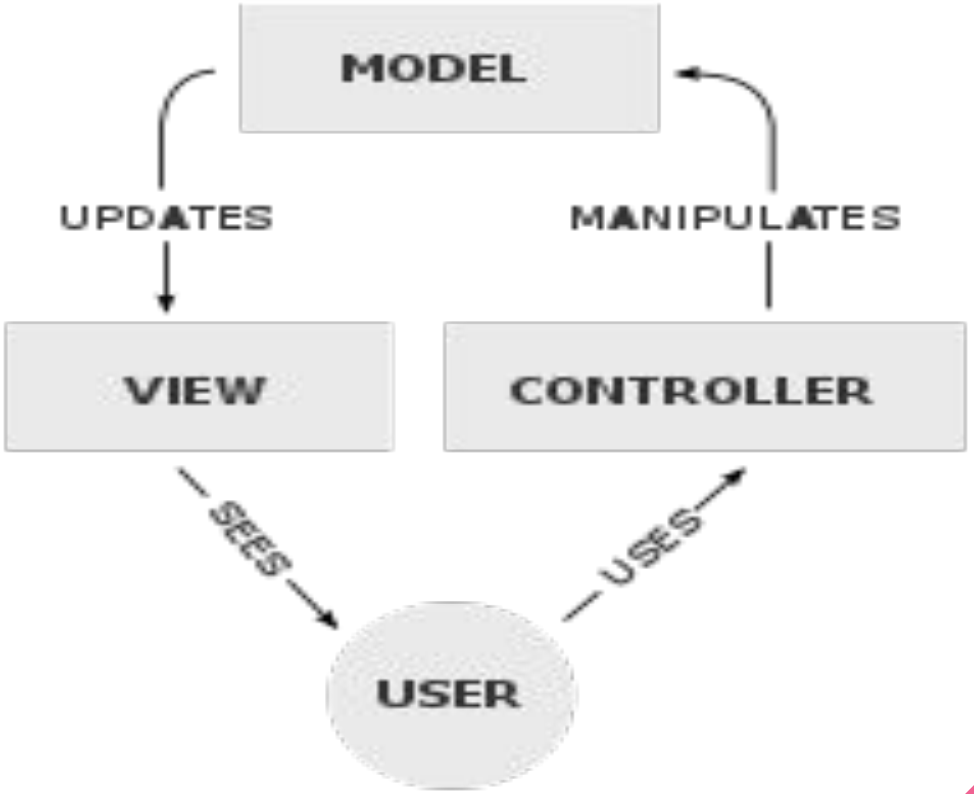
Continue...

The parts of MVC

- **Model:**
  - Model code typically reflects real-world things.
  - This code can hold raw data, or it will define the essential components of your app.

- **View:**
  - View code is made up of all the functions that directly interact with the user.
  - This is the code that makes your app look nice, and otherwise defines how your user sees and interacts with it.

- **Controller:**
  - Controller code acts as a liaison between the Model and the View, receiving user input and deciding what to do with it.
  - It's the brains of the application, and ties together the model and the view.


MODEL
UPDATES
MANIPULATES
VIEW
CONTROLLER
SEES
USES
USER

Continue...

MVC in the Real World

- MVC is helpful when planning your app, because it gives you an outline of how your ideas should be organized into actual code.
- For instance, let's imagine you're creating a To-do list app.
- The **Model** in a todo app might define what what a “task” is and that a “list” is a collection of tasks.
- The **View** code will define what the todos and lists looks like, visually. The tasks could have large font, or be a certain color.
- Finally, the **Controller** could define how a user adds a task, or marks another as complete. The Controller connects the View’s add button to the Model, so that when you click “add task,” the Model adds a new task.

Advantages of MVC

- Multiple developers can work simultaneously on the model, controller and views.
- MVC enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together.
- The very nature of the MVC framework is such that there is low coupling among models, views or controllers.
- *Ease of modification* – Because of the separation of responsibilities, future development or modification is easier

List of MVC Web Frameworks

- Zend for PHP
- Struts for Java
- Rails for Ruby
- Django for Python
- Merb for Ruby (for the experienced)
- ASP.NET MVC for .NET

Introduction to Ruby

- Ruby is an open-source and is freely available on the Web, but it is subject to a license.
- It is similar to Python and PERL.
- Ruby can be embedded into Hypertext Markup Language (HTML).
- 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.

Comparison b/w Ruby and Python

Terms	Ruby	Python
Definition	Ruby is an open source web application programming language	Python is a high level programming language.
Object-Oriented	Fully object oriented programming language.	Not fully object oriented programming language.
Mixins	Mixins are used.	Mixins can't be used.
Web frameworks	Ruby on Rails	Django
Usage	Apple, Github, Twitter etc.	Google, Instagram, Morzilla firefox etc..
Buitl-in classes	Built-in classes can be modified.	Built-in classes can't be modified.

Example

Example1.rb

```
length = gets.chomp.to_i
breadth = gets.chomp.to_i
area= length* breadth
puts " #{area}"
```

Input:
20
10

Output
200

Data Types

- Data types in ruby:
  - Numbers
  - Strings
  - Symbols
  - Hashes
  - Arrays
  - Booleans

Variables

- **Local Variables** − Local variables are the variables that are defined in a method.
- **Instance Variables** − Instance variables are available across methods for any particular instance or object. Preceded by the at sign (@).
- **Class Variables** − Class variables are available across different objects. Preceded by the sign @@.
- **Global Variables** −If you want to have a single variable, which is available across classes, you need to define a global variable. preceded by the dollar sign ($).

```ruby
#Example2.rb

class Square
        @@object_count = 0     #class variable

        def initialize(side)
                @side = side
                @@object_count = @@object_count+1
        end

        def get_area
                return @side*@side
        end

        def get_perimeter
                return 4*@side
        end

        def self.no_of_object       #class method
                print @@object_count
        end
end
```

```ruby
a = Square.new(4)
b = Square.new(16)
puts a.get_area
puts b.get_perimeter


puts Square.no_of_object   #calling class method
```

Strings

String literals are sequences of characters between single or double quotation marks.

Example3.rb

```
puts "Hello World"
puts 'Hello World'

puts 'I like' + ' Ruby'          # String concatenation

puts 'It\'s my Ruby'             # Escape sequence

puts 'Hello' * 3                 # Displays the string three times

PI = 3.1416
puts PI
```

Output

```
Hello World
Hello World
I like Ruby
It's my Ruby
HelloHelloHello
3.1416
```

Arrays

- An **Array** is just a list of items in order.
- Negative index values count from the end of the array, so the last element of an array can also be accessed with an index of -1.
- Reading an element beyond the end of an array (with an index >= **size**) or before the beginning of an array (with an index < **-size**), Ruby simply returns **nil** and does not throw an exception.
- Ruby's arrays are mutable - arrays are dynamically resizable; you can append elements to them and they grow as needed.

Example

```
# Example4.rb

var1 = []                    # Empty array
puts var1[0]                 # Array index starts from 0

fruit= 'mango'
var4 = [80.5, fruit, [true, false]]   #3 objects float, string, array
puts var4[2]

newarr = [45, 23, 1, 90]
puts newarr.sort
puts newarr.length
puts newarr.first
puts newarr.last

locations = ['Pune', 'Mumbai', 'Bangalore']
locations.each do |loc|
        puts loc
end
```

```
locations.delete('Mumbai')

locations.each do |loc|
        puts loc
end
```

Output

```
[true, false]
[1, 23, 45, 90]
4
45
90
Pune
Mumbai
Bangalore
Pune
Bangalore
```

```
#Example5.rb

def func1
        10.times do |num|
                square = num * num
                return num, square if num > 5
        end
end

# using parallel assignment to collect the return value

num, square = func1
puts num
puts square
```

Output

6
36

- The **times** method of the **Integer** class iterates block num times, passing in values from zero to num-1.
- If **return** statement, returns multiple parameters, the method returns them in an array. Parallel assignment is used to collect return value.

Parallel Assignment

- Once Ruby sees more than one rvalues(right hand side values) in an assignment, the rules of parallel assignment come into play.
- First, all the rvalues evaluated, left to right, and collected into an array (unless they are already an array).
- This array will be the eventual value returned by the overall assignment.
- Next, the left hand side (lhs) is inspected. If it contains a single element, the array is assigned to that element.
  - a = 1, 2, 3, 4 # => a == [1, 2, 3, 4]
  - b = [1, 2, 3, 4] # => b == [1, 2, 3, 4]
- If the lhs contains a comma, Ruby matches values on the rhs against successive elements on the lhs. Excess elements are discarded.
  - a, b = 1, 2, 3, 4 # => a == 1, b == 2
  - c, = 1, 2, 3, 4 # => c == 1

Symbols

- A symbol looks like a variable name but it's prefixed with a colon. Examples - **:action**, **:line_items**.
- They are useful because a given symbol name refers to the same object throughout a Ruby program.

- They are more efficient than strings. Two strings with the same contents are two different objects, but for any given name there is only one Symbol object. This can save both time and memory.

- When do we use string and symbol?
  - If the contents (the sequence of characters) of the object are important, use string.
  - If the identity of the object is important, use a symbol

#Example6.rb

```
puts "string".object_id
puts "string".object_id
puts :symbol.object_id
puts :symbol.object_id
```

Output

```
21508360
21508100
802268
802268
```

Hashes

Hashes (sometimes known as associative arrays, maps, or dictionaries) are similar to arrays and index can be a objects of any types: strings, regular expressions, and so on.

Hash stores two objects - the index (normally called the key) and the value. The values in a hash can be objects of any type.

#Example7.rb

```
sub = {'CSE' => 'DAA', 'ECE' => 'Signal Proces', 'Mech' => 'SOM'}

puts sub.length
puts sub['CSE']
puts sub
puts sub['Mech']
```

Output

```
3
DAA
{"CSE"=>"DAA", "ECE"=>"Signal Proces", "Mech"=>"SOM"}
SOM
```

Using Symbols as Hash Keys

```
#Examle8.rb

people = Hash.new
people[:nickname] = 'Abc'
people[:language] = 'English'
people[:lastname] = 'def'

puts people[:lastname]    #def
```

```
people = {:nickname=>'Abc', :language=> 'English', :lastname=>'def'}

Or

people = {nickname: 'Abc', language: 'English', lastname: 'def'}
```

Inheritance

- Ruby supports only *single class inheritance*, it does not support multiple class inheritance but it supports *mixins*. The *mixins* are designed to implement multiple inheritances in Ruby, but it only inherits the interface part.
- The benefit of inheritance is that classes lower down the hierarchy get the features of those higher up, but can also add specific features of their own.

Example

```
#Example8.rb

class Mammal
  def breathe
        puts "breathe..."
  end
end

class Cat < Mammal
  def speak
        puts "Meow"
  end
end

cat1 = Cat.new
cat1.breathe          #breathe...
Cat1.speak            #Meow
```

```
#Example9.rb   Method overriding

class Bird
        def preen
                puts "cleaning feathers."
        end
        def fly
                puts "flying."
        end
end
class Penguin < Bird
        def fly
                puts "Sorry. I'd rather swim."
        end
end

p = Penguin.new
p.preen
p.fly
```

Modules

- Module is a way of grouping methods, classes, and constants together. Benefits of using Module are
  - Modules provide a *namespace* and prevent name clashes.
  - Modules implement the *mixin* facility.
- Syntax

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

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

Example

```
#support.rb


module Week
  FIRST_DAY = "Sunday"

  def Week.weeks_in_month
    puts “4 weeks in a month"
  end

  def Week.weeks_in_year
    puts “52 weeks in a year"
  end

end
```

```
$LOAD_PATH << '.'
require "support"

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

d1 = Decade.new
puts Week::FIRST_DAY
Week.weeks_in_month
d1.no_of_months
```

Output

```
Sunday
You have four weeks in a month
Sunday
120
```

Mixins

- Ruby does not support multiple inheritance directly but Ruby Modules have another wonderful use.
- Modules eliminate the need for multiple inheritance, providing a facility called a *mixin*.
- Mixins provide a controlled way of adding functionality to classes.

Example

```
module A
  def a1
  end
  def a2
  end
end

module B
  def b1
  end
  def b2
  end
end

class Sample
        include A
        include B

        def s1
        end
end
```

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

- Module A consists of the methods a1 and a2. Module B consists of the methods b1 and b2.
- The class Sample includes both modules A and B and can access all four methods, namely, a1, a2, b1, and b2.
- Therefore, the class Sample inherits from both the module.
- This is known as multiple inheritance or a *mixin*.