rubyeffect^

Session #5 - Ruby
Classes, Objects

Topics

- Classes
  - Class
  - Class syntax
  - Class variables, methods
  - public, private, attr_reader, attr_writer, attr_accessor
- Objects
  - Objects
  - Create Object with new
  - Inheritance, override
- Introduction to modules and mixins

Class

All object oriented programming languages has classes and objects. And, ruby too has them. A class is the blueprint from which individual objects are created. A class is made up of a collection of variables representing internal state and methods providing behaviours that operate on that state.

- Typically, you create a new class by using *class* keyword followed by a name.
- 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
- The class definition may contain method, class variable, and instance variable declarations

Class syntax, Objects with ‘new’

Like said in the previous slide a class is defined with a *class* keyword followed by a name. Every class ends with *end* keyword.

```
class SampleClass
  # some code describing the class behavior
  def some_method
    # some code related to method
  end
end
object1 = SampleClass.new
object2 = SampleClass.new
```

Variables in 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.

Variables in Ruby Class(Contd.)

- 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 ($).

Variables in Ruby Class(Contd.)

Example of different variables in Ruby Class:

```
$types_of_cutomers = 4 # global variable
class Customer
  @@no_of_customers = 100 # class variable
   def initialize(id, name, addr) # local variables
      @customer_id = id # instance variable
      @customer_name = name # instance variable
      @customer_addr = addr # instance variable
   end
end
```

Attribute accessors

An object's instance variables are its attributes. They are used to read and write these instance variables from outside the class. We typically use a class object created with *new* keyword.

There are different types of accessor methods ruby provides:

- attr_reader
- attr_writer
- attr_accessor

Attribute accessors(Contd.)

Example:

```
class Fruit
  attr_accessor :kind
  attr_reader :color
  attr_writer :state
  def inspect
     @color = "yellow"
    "a fruit of the #{@kind} variety, #{@color} color and is also #{@state}"
  end
end
f1 = Fruit.new
f1.kind = "banana" ; f1.state = "fresh"
f1.kind ; f1.state ; #f1.color
p f1
```

Methods and access control in Classes

We've covered the basics of methods in previous section and we shall dive into access control over them. Ruby gives you three levels of protection:

- **Public** methods can be called by everyone - no access control is enforced. *A class's instance methods (these do not belong only to one object; instead, every instance of the class can call them) are public by default*; anyone can call them. The **initialize** method is always private.
- **Protected** methods can be invoked only by objects of the defining class and its subclasses. Access is kept within the family. However, usage of **protected** is limited.

Methods and access control in Classes(Contd.)

Example:

```
class ParentClass
  def parent_method_1          # this method is public
    p "parent method 1"
  end
  protected
    def parent_method_2    # this method is protected
      p "parent method 2"
    end
  private
    def parent_method_3  # this method is private
      p "parent method 3"
    end
end
```

Methods and access control in Classes(Contd.)

- **Private** methods cannot be called with an explicit receiver - the receiver is always **self**. This means that private methods can be called only in the context of the current object; you cannot invoke another object's private methods

Inheritance

Inheritance is a relation between two classes. We know that all cats are mammals, and all mammals are animals. 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. If all mammals breathe, then all cats breathe. Ruby supports single inheritance not multiple. Some other languages support multiple inheritance, a feature that allows classes to inherit features from multiple classes, but Ruby *doesn't* support this.

Inheritance(Contd.)

Example:

```
class Mammal
  def breathe
    puts "inhale and exhale"
  end
end
class Cat < Mammal
  def speak
    puts "Meow"
  end
end
cat = Cat.new
cat.breathe ; cat.speak
```

Method Overriding

Method overriding, in object oriented programming, is a language feature that allows a subclass to provide a specific implementation of a method that is already provided by one of its superclasses. The implementation in the subclass *overrides* (replaces) the implementation in the superclass.

Method Overriding(Contd.)

Example:

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

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

Intro to Modules and Mixins

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

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

Syntax:

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

References

- http://ruby-doc.org/core-2.2.3/Class.html
- http://www.rubyist.net/~slagell/ruby/accessors.html
- http://rubylearning.com/satishtalim/ruby_access_control.html
- http://www.tutorialspoint.com/ruby/ruby_classes.htm
- https://en.wikibooks.org/wiki/Ruby_Programming/Syntax/Classes
- http://rubylearning.com/satishtalim/ruby_overriding_methods.html
- http://www.tutorialspoint.com/ruby/ruby_modules.htm