Classes and Objects in Ruby

Kuhoo Gupta

Computer Science and Engineering
IIT Bombay
kuhoo@cse.iitb.ac.in

Nov 29, 2004

Classes and Objects

Rubby Object

A set of flags, instance variables and an associated class

Rubby Class

An object of class *Class*. Contains objects, a list of methods and a reference to a superclass

Method Invocation

Looks at the list of methods in the receiver's class. If it doesn't find the method there, it looks in the superclass, and then in the superclass's superclass, and so on.

Class Definition

- Defines the new class. The names is identifier beginning with uppercase character

Syntax & Example

```
    class identifier ['<' superclass ]
        expr..
    end
Example     class Foo < Super
        def test . . .
        end
            :
    end
```

Method Definition

- Defines the new method. Name should be either identifier or re-definable operators (e.g. ==, +, -, etc.)

Syntax & Example

```
    def method_name ['(' [arg ['=' default]]...[',' '*' arg ]')']
        expr..
    end
Example      def fact(n)
        if n == 1 then ; 1
        else ; n * fact(n-1)
        end
    end
```

Object-Specific Classes

- Class tied to a particular Object

Example

```
a = "hello" ; b = a.dup
class <<a
    def to_s ; "The value is '#self'" ; end
    def twoTimes ; self + self ; end
end    a.to_s >> "The value is 'hello'"
a.twoTimes >> "hellohello"
b.to_s >> "hello"
```

Mixin Modules

- When a class includes a module, that module's instance methods become available as instance methods of the class.

Example

```
    module SillyModule
        def hello ; "Hello." ; end
    end
    class SillyClass
        include SillyModule
    end
s = SillyClass.new ; s.hello >> "Hello."
```

Extending Objects

- Mixing a module into an object using *extend*

Example

```
module Humor
    def tickle
        "hee, hee!"
    end
end
a = "Grouchy"
a.extend Humor
a.tickle >> "hee, hee!"
```

Classes & Modules

- In C++ and Java, class definitions are processed at compile time
- In Ruby, class and module definitions are executable code
- Although definitions are parsed at compile time, the classes and modules are created at runtime, when the definition is encountered
- Structures your programs far more dynamically than in most conventional languages
- You can make decisions once, when the class is being defined, rather than each time that objects of the class are used

Dynamic Support for Definining Routines

- The class in the following example decides as it is being defined what version of a decryption routine to create.

Example

```
class MediaPlayer
  include Tracing if $DEBUGGING
  if ::EXPORT_VERSION
    def decrypt(stream)
      raise "Decryption not available"
    end
  else ; def decrypt(stream) ; # ... end
  end
end
```

Class Definitions as Executable

Example

```
class Test
  puts "Type of self = #{self.type}"
  puts "Name of self = #{self.name}"
end
```

produces

```
Type of self = Class
Name of self = Test
```

Class as Current Object

Example

```
class Test
  def Test.sayHello
    puts "Hello from #name"
  end
  sayHello
end
```

produces

```
Hello from Test
```

Explanation of the Previous Example

- We define a class method, Test.sayHello, and then call it in the body of the class definition
- Within sayHello, we call name, an instance method of class Module
- Because Module is an ancestor of Class, its instance methods can be called without an explicit receiver within a class definition

Class Names are Constants

- When we invoke a class method, we are sending a message to the Class object itself
- When we say something such as `String.new("gumby")`, we are sending the message new to the object that is class String
- The receiver of a message should be an object reference, which implies that there must be a constant called "String" somewhere containing a reference to the String object
- All the built-in classes, along with the classes we define, have a corresponding global constant with the same name as the class

Class as Other Ruby Objects

- The fact that class names are just constants means that you can treat classes just like any other Ruby object
- We can copy them, pass them to methods, and use them in expressions

```
def factory(klass, *args); klass.new(*args); end
factory(String, "Hello") >> "Hello"
factory(Dir, ".") >> #<Dir:0x401b3234>
flag = true
(flag ? Array : Hash)[1, 2, 3, 4] >> [1, 2, 3, 4]
flag = false
(flag ? Array : Hash)[1, 2, 3, 4] >> {1=>2, 3=>4}
```

Thank You

THANK YOU