Ruby Programming/Syntax/Classes

Classes are the basic template from which object instances are created. A class is made up of a collection of variables representing internal state and methods providing behaviours that operate on that state.

Contents

- 1 Class Definition
  - 1.1 Instance Variables
  - 1.2 Accessor Methods
  - 1.3 Class Variables
  - 1.4 Class Instance Variables
  - 1.5 Class Methods
  - 1.6 Instantiation
- 2 Declaring Visibility
  - 2.1 Private
  - 2.2 Public
  - 2.3 Protected
  - 2.4 Instance Variables
- 3 Inheritance
- 4 Mixing in Modules
- 5 Ruby Class Meta-Model

Class Definition

Classes are defined in Ruby using the `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 (CamelCase). The class definition may contain method, class variable, and instance variable declarations as well as calls to methods that execute in the class context at read time, such as attr_accessor. The class declaration is terminated by the `end` keyword.

Example:

```
class MyClass
  def some_method
  end
end
```

Instance Variables

Instance variables are created for each class instance and are accessible only within that instance. They are accessed using the @ operator. Outside of the class definition, the value of an instance variable can only be read or modified via that instance's public methods.

Example:

```ruby
class MyClass
  @one = 1
  def do_something
    @one = 2
  end
  def output
    puts @one
  end
end
instance = MyClass.new
instance.output
instance.do_something
instance.output
```

Surprisingly, this outputs:

```
nil
2
```

This happens (nil in the first output line) because *@one* defined below *class MyClass* is an instance variable belonging to the class object (note this is not the same as a class variable and could not be referred to as *@@one*), whereas *@one* defined inside the *do_something* method is an instance variable belonging to instances of *MyClass*. They are two distinct variables and the first is accessible only in a class method.

Accessor Methods

As noted in the previous section, an instance variable can only be directly accessed or modified within an instance method definition. If you want to provide access to it from outside, you need to define public accessor methods, for example

```ruby
class MyClass
  def initialize
    @foo = 28
  end

  def foo
    return @foo
  end

  def foo=(value)
    @foo = value
  end
end

instance = MyClass.new
puts instance.foo
instance.foo = 496
puts instance.foo
```

Note that ruby provides a bit of syntactic sugar to make it look like you are getting and setting a variable directly; under the hood

```ruby
a = instance.foo
instance.foo = b
```

are calls to the foo and foo= methods

```
a = instance.foo()
instance.foo=(b)
```

Since this is such a common use case, there is also a convenience method to autogenerate these getters and setters:

```
class MyClass
  attr_accessor :foo

  def initialize
    @foo = 28
  end
end

instance = MyClass.new
puts instance.foo
instance.foo = 496
puts instance.foo
```

does the same thing as the above program. The attr_accessor method is run at read time, when ruby is constructing the class object, and it generates the foo and foo= methods.

However, there is no requirement for the accessor methods to simply transparently access the instance variable. For example, we could ensure that all values are rounded before being stored in foo:

```
class MyClass
  def initialize
    @foo = 28
  end

  def foo
    return @foo
  end

  def foo=(value)
    @foo = value.round
  end
end

instance = MyClass.new
puts instance.foo
instance.foo = 496.2
puts instance.foo #=> 496
```

Class Variables

Class variables are accessed using the @@ operator. These variables are associated with the class hierarchy rather than any object instance of the class and are the same across all object instances. (These are similar to class "static" variables in Java or C++).

Example:

```
class MyClass
  @@value = 1
  def add_one
    @@value= @@value + 1
  end

  def value
    @@value
  end
end
```

```
instanceOne = MyClass.new
instanceTwo = MyClass.new
puts instanceOne.value
instanceOne.add_one
puts instanceOne.value
puts instanceTwo.value
```

Outputs:

```
1
2
2
```

Class Instance Variables

Classes can have instance variables. This gives each class a variable that is not shared by other classes in the inheritance chain.

```
class Employee
  class << self; attr_accessor :instances; end
  def store
    self.class.instances ||= []
    self.class.instances << self
  end
  def initialize name
    @name = name
  end
end
class Overhead < Employee; end
class Programmer < Employee; end
Overhead.new('Martin').store
Overhead.new('Roy').store
Programmer.new('Erik').store
puts Overhead.instances.size    # => 2
puts Programmer.instances.size  # => 1
```

For more details, see MF Bliki: ClassInstanceVariables (http://martinfowler.com/bliki/ClassInstanceVariable.html)

Class Methods

Class methods are declared the same way as normal methods, except that they are prefixed by `self`, or the class name, followed by a period. These methods are executed at the Class level and may be called without an object instance. They cannot access instance variables but do have access to class variables.

Example:

```
class MyClass
  def self.some_method
    puts 'something'
  end
end
MyClass.some_method
```

Outputs:

```
something
```

Instantiation

An object instance is created from a class through the a process called *instantiation*. In Ruby this takes place through the Class method `new`.

Example:

```
anObject = MyClass.new(parameters)
```

This function sets up the object in memory and then delegates control to the initialize function of the class if it is present. Parameters passed to the new function are passed into the `initialize` function.

```
class MyClass
  def initialize(parameters)
  end
end
```

Declaring Visibility

By default, all methods in Ruby classes are public - accessible by anyone. There are, nonetheless, only two exceptions for this rule: the global methods defined under the Object class, and the initialize method for any class. Both of them are implicitly private.

If desired, the access for methods can be restricted by public, private, protected object methods.

It is interesting that these are not actually keywords, but actual methods that operate on the class, dynamically altering the visibility of the methods, and as a result, these 'keywords' influence the visibility of all following declarations until a new visibility is set or the end of the declaration-body is reached.

Private

Simple example:

```
class Example
  def methodA
  end

  private # all methods that follow will be made private: not accessible for outside objects

  def methodP
  end
end
```

If private is invoked without arguments, it sets access to private for all subsequent methods. It can also be invoked with named arguments.

Named private method example:

```
class Example
  def methodA
  end

  def methodP
  end
```

```
  private :methodP
end
```

Here private was invoked with an argument, altering the visibility of methodP to private.

Note for class methods (those that are declared using def ClassName.method_name), you need to use another function: private_class_method

Common usage of private_class_method is to make the "new" method (constructor) inaccessible, to force access to an object through some getter function. A typical Singleton implementation is an obvious example.

```
class SingletonLike
  private_class_method :new

  def SingletonLike.create(*args, &block)
    @@inst = new(*args, &block) unless @@inst
    return @@inst
  end
end
```

Note : another popular way to code the same declaration

```
class SingletonLike
  private_class_method :new

  def SingletonLike.create(*args, &block)
    @@inst ||= new(*args, &block)
  end
end
```

More info about the difference between C++ and Ruby private/protected: http://lylejohnson.name/blog/?p=5

One person summed up the distinctions by saying that in C++, “private” means “private to this class”, while in Ruby it means “private to this instance”. What this means, in C++ from code in class A, you can access any private method for any other object of type A. In Ruby, you cannot: you can only access private methods for your instance of object, and not for any other object instance (of class A).

Ruby folks keep saying "private means you cannot specify the receiver". What they are saying, if method is private, in your code you can say:

```
class AccessPrivate
  def a
  end
  private :a # a is private method

  def accessing_private
    a              # sure!
    self.a         # nope! private methods cannot be called with an explicit receiver at all, even if that receiver is "self
    other_object.a # nope, a is private, you can't get it (but if it was protected, you could!)
  end
end
```

Here, "other_object" is the "receiver" that method "a" is invoked on. For private methods, it does not work. However, that is what "protected" visibility will allow.

Public

Public is default accessibility level for class methods. I am not sure why this is specified - maybe for completeness, maybe so that you could dynamically make some method private at some point, and later - public.

In Ruby, visibility is completely dynamic. You can change method visibility at runtime!

Protected

Now, “protected” deserves more discussion. Those of you coming from Java or C++ have learned that in those languages, if a method is “private”, its visibility is restricted to the declaring class, and if the method is “protected”, it will be accessible to children of the class (classes that inherit from parent) or other classes in that package.

In Ruby, “private” visibility is similar to what “protected” is in Java. Private methods in Ruby are accessible from children. You can’t have truly private methods in Ruby; you can’t completely hide a method.

The difference between protected and private is subtle. If a method is protected, it may be called by any instance of the defining class or its subclasses. If a method is private, it may be called only within the context of the calling object---it is never possible to access another object instance's private methods directly, even if the object is of the same class as the caller. For protected methods, they are accessible from objects of the same class (or children).

So, from within an object "a1" (an instance of Class A), you can call private methods only for instance of "a1" (self). And you cannot call private methods of object "a2" (that also is of class A) - they are private to a2. But you can call protected methods of object "a2" since objects a1 and a2 are both of class A.

Ruby FAQ (http://www.rubycentral.com/faq/rubyfaq-7.html) gives following example - implementing an operator that compares one internal variable with a variable from the same class (for purposes of comparing the objects):

```
def <=>(other)
  self.age <=> other.age
end
```

If age is private, this method will not work, because other.age is not accessible. If "age" is protected, this will work fine, because self and other are of same class, and can access each other's protected methods.

To think of this, protected actually reminds me of the "internal" accessibility modifier in C# or "default" accessibility in Java (when no accessibility keword is set on method or variable): method is accessible just as "public", but only for classes inside the same package.

Instance Variables

Note that object instance variables are not really private, you just can't see them. To access an instance variable, you need to create a getter and setter.

Like this (no, don't do this by hand! See below):

```
class GotAccessor
  def initialize(size)
    @size = size
  end

  def size
    @size
  end
  def size=(val)
    @size = val
```

```ruby
  end
end

# you could the access @size variable as
# a = GotAccessor.new(5)
# x = a.size
# a.size = y
```

Luckily, we have special functions to do just that: attr_accessor, attr_reader, attr_writer. attr_accessor will give you get/set functionality, reader will give only getter and writer will give only setter.

Now reduced to:

```ruby
class GotAccessor
  def initialize(size)
    @size = size
  end

  attr_accessor :size
end

# attr_accessor generates variable @size accessor methods automatically:
# a = GotAccessor.new(5)
# x = a.size
# a.size = y
```

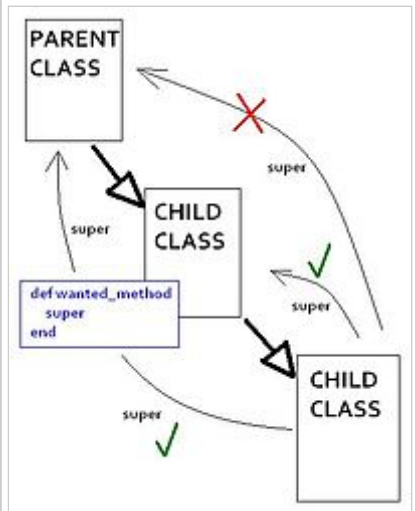
Inheritance

A class can *inherit* functionality and variables from a *superclass*, sometimes referred to as a *parent class* or *base class*. Ruby does not support *multiple inheritance* and so a class in Ruby can have only one superclass. The syntax is as follows:

```ruby
class ParentClass
  def a_method
    puts 'b'
  end
end

class SomeClass < ParentClass  # < means inherit (or "extends" if you are from Java
  def another_method
    puts 'a'
  end
end

instance = SomeClass.new
instance.another_method
instance.a_method
```



The *super* keyword only accessing the direct parents method. There is a workaround though.

Outputs:

```
a
b
```

All non-private variables and functions are inherited by the child class from the superclass.

If your class overrides a method from parent class (superclass), you still can access the parent's method by using 'super' keyword.

```
class ParentClass
  def a_method
    puts 'b'
  end
end

class SomeClass < ParentClass
  def a_method
    super
    puts 'a'
  end
end

instance = SomeClass.new
instance.a_method
```

Outputs:

```
b
a
```

(because a_method also did invoke the method from parent class).

If you have a deep inheritance line, and still want to access some parent class (superclass) methods directly, you can't. *super* only gets you a direct parent's method. But there is a workaround! When inheriting from a class, you can alias parent class method to a different name. Then you can access methods by alias.

```
class X
  def foo
    "hello"
  end
end

class Y < X
  alias xFoo foo
  def foo
    xFoo + "y"
  end
end

class Z < Y
  def foo
    xFoo + "z"
  end
end

puts X.new.foo
puts Y.new.foo
puts Z.new.foo
```

Outputs

```
hello
helloy
helloz
```

Mixing in Modules

First, you need to read up on modules Ruby modules (http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut_modules.html). Modules are a way of grouping together some functions and variables and classes, somewhat like classes, but more like namespaces. So a module is not really a class. You can't instantiate a Module, and a module cannot use Self to refer to itself. Modules can have module methods (as classes can have class methods) and instances methods as well.

We can, though, include the module into a class. Mix it in, so to speak.

```
module A
  def a1
    puts 'a1 is called'
  end
end

module B
  def b1
    puts 'b1 is called'
  end
end

module C
  def c1
    puts 'c1 is called'
  end
end

class Test
  include A
  include B
  include C

  def display
    puts 'Modules are included'
  end
end

object=Test.new
object.display
object.a1
object.b1
object.c1
```

Outputs:

```
Modules are included
a1 is called
b1 is called
c1 is called
```

The code shows Multiple Inheritance using modules.

Ruby Class Meta-Model

In keeping with the Ruby principle that everything is an object, classes are themselves instances of the class Class. They are stored in constants under the scope of the module in which they are declared. A call to a method on an object instance is delegated to a variable inside the object that contains a reference to the class of that object. The method implementation exists on the Class instance object itself. Class methods are implemented on meta-classes that are linked to the existing class instance objects in the same way that those classes instances are linked to them. These meta-classes are hidden from most Ruby functions.