Ruby Inheritance

<Open Classes | TOC | Overriding Methods>

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. *In Ruby, a class can only inherit from a single other class*. Some other languages support multiple inheritance, a feature that allows classes to inherit features from multiple classes, but Ruby *doesn't* support this.

We can express this concept in Ruby - see the **p033mammal.rb** program below:

```
class Mammal
  def breathe
    puts "inhale and exhale"
  end
end

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

rani = Cat.new
rani.breathe
rani.speak
```

Though we didn't specify how a Cat should breathe, every cat will inherit that behaviour from the Mammal class since Cat was defined as a subclass of Mammal. (In OO terminology, the smaller class is a subclass and the larger class is a super-class. The subclass is sometimes also known as a derived or child class and the super-class as base or parent class). Hence from a programmer's standpoint, cats get the ability to breathe for free; after we add a speak method, our cats can both breathe and speak.

There will be situations where certain properties of the super-class should not be inherited by a particular subclass. Though birds generally know how to fly, penguins are a flightless subclass of birds. In the example **p034bird.rb** below, we **override** fly in class Penguin:

```
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
```

Rather than exhaustively define every characteristic of every new class, we need only to append or to redefine the differences between each subclass and its super-class. This use of inheritance is sometimes called differential programming. It is one of the benefits of object-oriented programming.

The above two programs are taken from the online **Ruby User's Guide**.

Thus, **Inheritance** allows you to create a class that is a refinement or specialization of another class. Inheritance is

indicated with <.

Here's another example, **p035inherit.rb**

```
           class GF
 def initialize
  puts 'In GF class'
 end
 def gfmethod
  puts 'GF method call'
 end
end

# class F sub-class of GF
class F < GF
 def initialize
  puts 'In F class'
 end
end

# class S sub-class of F
class S < F
 def initialize
  puts 'In S class'
 end
end
son = S.new
son.gfmethod
```

A class can only inherit from one class at a time (i.e. a class can inherit from a class that inherits from another class which inherits from another class, but a single class can not inherit from many classes at once).

There are many classes and modules (more on this later) built into the standard Ruby language. They are available to every Ruby program automatically; no **require** is required. Some built-in classes are **Array, Bignum, Class, Dir, Exception, File, Fixnum, Float, Integer, IO, Module, Numeric, Object, Range, String, Thread, Time**. Some built-in modules are **Comparable, Enumerable, GC, Kernel, Math**.

The **BasicObject** class is the parent class of all classes in Ruby. Its methods are therefore available to all objects unless explicitly overridden. Prior to Ruby 1.9, **Object** class was the root of the class hierarchy. The new class **BasicObject** serves that purpose, and **Object** is a subclass of **BasicObject**. **BasicObject** is a very simple class, with almost no methods of its own. When you create a class in Ruby, you extend **Object** unless you explicitly specify the super-class, and most programmers will never need to use or extend **BasicObject**.

In Ruby, **initialize** is an ordinary method and is inherited like any other.

IN RAILS: Inheritance is one of the key organizational techniques for Rails program design and the design of the Rails framework.

Inheritance and Instance Variables

Consider the code:

```
class Dog
  def initialize(breed)
    @breed = breed
  end
end

class Lab < Dog
  def initialize(breed, name)
    super(breed)
    @name = name
  end

  def to_s
    "(#@breed, #@name)"
  end
end

puts Lab.new("Labrador", "Benzy").to_s
```

When you invoke **super** with arguments, Ruby sends a message to the parent of the current object, asking it to

invoke a method of the same name as the method invoking **super**. **super** sends exactly those arguments.

The **to_s** method in class **Lab** references **@breed** variable from the super-class **Dog**. This code works as you probably expect it to:

```
puts Lab.new("Labrador", "Benzy").to_s ==> (Labrador, Benzy)
```

Because this code behaves as expected, you may be tempted to say that these variables are inherited. *That is not how Ruby works*.

All Ruby objects have a set of instance variables. These are **not** defined by the objects's class - they are simply created when a value is assigned to them. *Because instance variables are not defined by a class, they are unrelated to subclassing and the inheritance mechanism*.

In the above code, **Lab** defines an **initialize** method that chains to the **initialize** method of its super-class. The chained method assigns values to the variable **@breed**, which makes those variables come into existence for a particular instance of **Lab**.

The reason that they sometimes appear to be inherited is that instance variables are created by the methods that first assign values to them, and those *methods* are often inherited or chained.

Since instance variables have nothing to do with inheritance, it follows that an instance variable used by a subclass cannot "shadow" an instance variable in the super-class. If a subclass uses an instance variable with the same name as a variable used by one of its ancestors, it will overwrite the value of its ancestor's variable.

<Open Classes | TOC | Overriding Methods>