Lab 4: Class Inheritance and Modules in Ruby

Learning Objectives:

1. Inheritance in Ruby
2. Mixing in Modules
3. Inheritance Vs Modules
4. Private, Protected, and Public

Supporting files:

1. Inheritance_example.zip

1. Inheritance in Ruby

Inheritance is a feature of object oriented programming languages, that is used to indicate that if a newly created class will get most or all the features from another class. In Ruby, inheritance is implicitly defined when we create a class with a syntax such as ‘Class Foo < Bar’. This specific statement says "Create a class Foo that inherits from Bar." When we do this, any action that is taken on the instances of Foo seem as if they were taken on the instances of Bar. Here the Foo class becomes a child of the Bar class and the Bar class acts as the parent to Foo. This relationship lets us put the common functionalities in the parent class, then specialize these functionalities in the all the children classes of this parent.

When related with the inheritance relationship, there are three ways that the parent and child classes can interact:

- Actions on the instances of the child class imply the same actions on the parent.
- Actions on the instances of the child can override the actions on the parent.
- Modifications of the same functionality in the child can alter the actions on the parent.

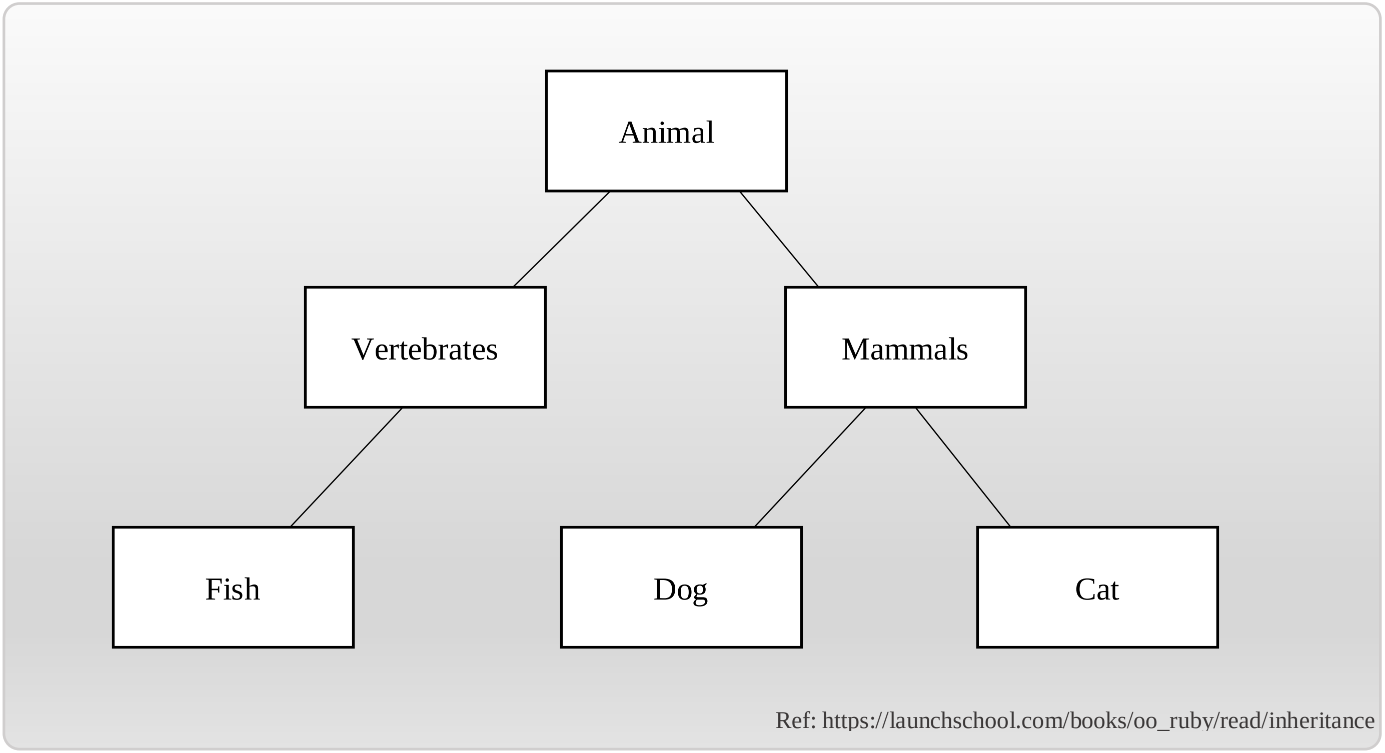
For example:

```
class Animal
  def speak
          "Grrrrrr…."
  end
end

class Dog < Animal
   def speak
          super + "Woof! Woof!"
  end
end

class Cat < Animal
end

casper = Dog.new
kitty = Cat.new
puts casper.speak         # => Output: Grrrrrr…. Woof! Woof!
puts kitty.speak                    # => Output: Grrrrrr….
```

In the above example, the symbol ‘<’ is used to signify that the Cat and Dog classes are children of the Animal class. While the Dog class overrides the speak method defined in its parent, the Cat class chooses to execute its parent’s speak method when needed. The Dog class also uses the *super* keyword in the speak method that allows it to call methods up in the hierarchy. In this case, the speak method definition of the Animal class.

2. Mixing in Modules

The purpose of inheritance in any OOP language is to have a clean code and put the common methods in one place. However, sometimes the situation can be more complicated than a simple inheritance relationship. For example, let us look at the following relationships:


Animal
Vertebrates
Mammals
Fish
Dog
Cat
Ref: https://launchschool.com/books/oo_ruby/read/inheritance

The above diagram shows what a pure class based inheritance looks like. Remember the goal of this is to put the right behavior (i.e., methods) in the right class so we don't need to repeat code in multiple classes. We can imagine that all Fish objects are related to Animals that are Vertebrates, so perhaps a swim method should be in the Vertebrates class. We can also imagine that all Mammal objects will have warm blood, so we can create a method called warm_blooded? in the Mammal class and have it return true. Therefore, the Cat and Dog objects will automatically inherit the warm_blooded? method from Mammal, but not the methods in Fish.

This type of hierarchical modeling works to some extent, but there are always exceptions. For example, we put a swim method in the Fish class, but some mammals can swim as well. We don't want to move the swim method into Animal because not all animals swim, and we don't want to create another swim method in Dog because that violates the DRY principle.

For concerns such as these, in Ruby we use modules. We group the necessary functionalities into a module and then mix in that module to the classes that require those behaviors. Here's an example:

```
module CanSwim
  def swim
    "I can swim!"
  end
end

class Animal; end

class Vertebrate < Animal; end

class Fish < Vertebrate
  include CanSwim        # mixing in CanSwim module
end

class Mammal < Animal; end

class Cat < Mammal; end

class Dog < Mammal
  include CanSwim        # mixing in CanSwim module
end
```

3. Inheritance Vs. Modules

Now that we know the two primary ways that Ruby implements inheritance, class inheritance and mixings in modules, we may wonder when to use one or the other. Here are a few things to remember when evaluating these two choices.

- We can only subclass from one class. But we can mix in as many modules as we wouldlike.
- If it's an "is-a" relationship, we choose class inheritance. If it's a "has-a" relationship, we choose modules. Example: a dog "is an" animal; a dog "has an" ability to swim.
- We cannot instantiate modules (i.e., no object can be created from a module). Modules are used only for namespacing and grouping common methods together.

4. Private, Protected, and Public

Finally, let us discuss something that is quite simple, but necessary. Right now, all the methods in our Dog class are public methods. A **public** method is a method that is available to anyone who is aware of the class. These methods are readily available for the rest of the program to use and comprise the class's interface.

Sometimes, you'll have methods that are doing work in the class but don't need to be available to the rest of the program. These methods can be defined as **private**. To define private methods, we use the reserved keyword private in our program and unless another reserved keyword is placed after it to negate it, anything below it is private. For example, in our Dog class we have one operation that takes place that we could move into a private method. When we initialize an object, we calculate the dog's age in human years. For example:

```
class Dog
  AGE = 7

  attr_accessor :name, :age

  def initialize(n, a)
    self.name = n
    self.age = a
  end

  private

  def age_in_years
    age * AGE
  end
end

casper = Dog.new(“Casper”, 4)
casper.age_in_years
```

Ref: https://launchschool.com/books/oo_ruby/read/inheritance

Output: gives an error:

```
        NoMethodError: private method `age_in_years' called for
#<Dog:0x007f8f431441f8 @name="Casper", @age=4>
```

Private methods are only accessible from other methods in the class. Hence, in order to fix this error, we cannot use self.age_in_years, because the age_in_years method is private. Remember that self.age_in_years is equivalent to casper.age_in_years, which is not allowed for private methods. Therefore, we have to just use #{age_in_years}. For example:

```
def age_disclosure
  "#{self.name}is #{age_in_years}" old in human years”
end
```

Ref: https://launchschool.com/books/oo_ruby/read/inheritance

Public and private methods are most common, but in some less common situations, we'll want an in-between approach that is protected methods. We can use the **protected** keyword to create protected methods. The easiest way to understand protected methods is to follow these two rules:

- from outside the class, protected methods act just like private methods.
- from inside the class, protected methods are accessible just like public methods.

For example,

```
class Animal
  def a_public_method
    "Will this work? " + self.a_protected_method
  end

  protected

  def a_protected_method
    "Yes, I'm protected!"
  end
end

fido = Animal.new
fido.a_public_method
```

Ref: https://launchschool.com/books/oo_ruby/read/inheritance

Output: Gives us the following error:

```
fido.a_protected_method
 # => NoMethodError: protected method `a_protected_method' called for
#<Animal:0x007fb174157110>
```

This demonstrates the second rule that we have discussed above. As per the rule, we can't call protected methods from outside of the class. The two rules for protected methods apply within the context of inheritance as well.