Web Application Architectures

Module 4: The Ruby Programming Language
Lecture 2: Classes and Inheritance

THE UNIVERSITY *of* NEW MEXICO

Classes

- Classes are defined using the keyword `class` followed by the name of the class. The name must begin with a capital, and the convention is to use CamelCase.

- To define a method, use the keyword `def`:
  Ex.

```
class MyClass
  @boo          # an instance variable
  def my_method
    @foo = 2    # an instance variable
  end
end

> mc = MyClass.new   # create a MyClass object
> mc.my_method       # => 2
> mc.boo             # => error
```

Methods

An instance variable can only be directly accessed or modified within a method definition.

Ex.

```
class MyClass
  def boo  # a getter method
    return @boo
  end
  def boo=(val)  # setter method
    @boo = val
  end
end

> mc = MyClass.new  # create a MyClass object
> boo = 1    # => 1
> boo        # => 1
```

Methods

- Notice that there is no return value specified in the methods above. Ruby methods have implicit return values – the value of the last expression executed in a method is its return value.

- The `return` statement still exists, but you don't need to use it.

  Ex.

  ```
  def min(x,y)
    if x < y then x else y end
  end
  ```

- When invoking a method, parentheses are optional.

Class Methods

- Class methods are created in the same way as normal methods, except they are prefixed by the keyword `self`.

Ex.

```
class MyClass
  def self.cls_method
    "MyClass type"
  end
end

> MyClass.cls_method  # => "MyClass type"
```

Methods

In Ruby the last character of a method name is often used to indicate its behavior:

- If the method ends with a question mark it indicates that the return value is boolean.
- If the method ends with an exclamation, it indicates that the method can change the state of the object.
- In the previous case, it is common to also provide a non-exclamation version of the method, which indicates that the modifies a copy of the object.
- The keyword `self` can be used inside an object's methods in order to refer to the current object.

Inheritance, Mixins and Extending Classes

- Only single inheritance is supported; however, the mixin capability associated with modules basically gives you multiple inheritance.
- Classes are never closed, you can always add methods to an existing class.
  - This applies to the classes you write as well as the standard, built-in classes.
  - You simply open up a class definition for an existing class, and the new contents you specify will be added to whatever's already defined for that class.

  Ex.

```
class Fixnum
  def previous
    return self-1
  end
end
```

Specifying Access

- Within a class definition you may specify access levels using the keywords `public`, `private` and `protected`.

- The behavior is a little different than in C++ or Java:
  - `public` – no access control, can be called by anyone.
  - `protected` – can be invoked only by objects of the defining class and its subclasses.
  - `private` – can only be called in the context of the current object, without on object reference on the LHS, i.e, two objects of the same class cannot invoke each others private methods. Thus, the receiver of a private method is always `self`.

- By default, every method in a class is `public`, and every instance variable is `protected`.

Accessors

- There is a shorthand way of providing accessors for an object's attributes:

```
      class Person
        attr_accessor :first_name, :last_name
      end
```

  will create attributes (instance variables) for `first_name` and `last_name`, as well as getter and setter methods for each.

- If you only want a getter method, use `attr_reader`, and if you only want a setter, use `attr_writer`

Inheritance

- The syntax for inheritance is:

```
class NewClass < SuperClass
   ...
end
```

- The `initialize` method, which is always private, is used to create a constructor that is invoked by calling `new` on a class name. E.g., `a = Array.new`
- You can create a module with its own namespace by using the keyword `module`, and include a number of classes within it. You can include a module within another program by using the keywork `require`, e.g., `require 'module_name'`
- Within a class, you use the keyword `include` to mixin a module. This makes all of the methods defined in that module a part of the class that includes the module.