Ministry of Education of Republic of Moldova

Technical University of Moldova

CIM Faculty

Anglophone Department

Report

on APPOO

Laboratory Work #1

Performed by: *st. gr. FAF-131,* *Ghidin Oxana*

Verified by: *Drahnea Vlad*

Chişinău, 2016

Task 0: Chose two languages, at least one of these to be Object Oriented (OO). Research and demonstrate in examples (code) how we can use (or simulate if there is no such possibility) core OOP concepts - inheritance, polymorphism, encapsulation.

Task 1: Make a free form comparison/analysis of these concepts in your two languages. For example you can use a grid with pros and cons. Max one A4 page. Push your report to public repo and submit link to it.

Swift

1. Inheritance [2]

Swift is a single-inheritance language.
"Child" classes, or ***subclasses***, inherit all the characteristics of their "parent" classes, or ***superclasses***.

See the following example

```
class Car : Vehicle {
  override init() {
    super.init()
    numberOfWheels = 4
  }
}
```

Car derives from the **Vehicle** class and inherits all of its methods and properties. It makes perfect sense to say, "A **Car** is-a **Vehicle**". If you can naturally say, "Subclass is-a Superclass", derivation usually makes sense.
Since there are provided values for all of the properties in the superclass, there's no need to explicitly set each in the subclass. For **Car**s, the **numberOfWheels** will be four. To express this **override** the initializer, call the superclass initializer, and then customize the **numberOfWheels** property to four. The order is significant. If there is called **super.init()** last, **numberOfWheels** would get reset back to 0. Fortunately, the Swift complier will not allow to make this mistake.

In ***Car.swift***, above your initializer, there are added

Ruby

1. Inheritance [1]

Inheritance is a relation between two classes. A child class inherits all the features of its parent class. Methods from the parent can be overridden in the child and new logic can be added.

Usually, inheritance is used to specialize a class. See the following example

```
class Document
  def initialize; end
  # logic to deal with any document
def print
    # logic to print any kind of document
  end
end
```

—

```
1 class XmlDocument < Document
2   # logic to deal with any document
3
4   def print
5     # logic to print a xml document
6   end
7 end
8
```

In Ruby it can be replicated a certain form of multi-inheritance through the use of modules as mix-ins :

additional properties:

```
var isConvertible:Bool = false
var isHatchback:Bool = false
var hasSunroof:Bool = false
var numberOfDoors:Int = 0
```

Now, any ***Car*** object can be customized with all properties from the ***Vehicle*** and ***Car*** classes

```
1  module Presenter
2    def to_html; end
3  end
```

—

```
1  class XmlDocument < Document
2    include Presenter
3    # can call the method to_html
4  end
5
```

2. Polymorphism[3]

Whenever you have declared a variable of a particular type, you have always stored an object of that exact same type into the variable. For example, the following code declares a variable of type **UITextField**, and then creates an instance of **UITextField** and stores it in the **textField** variable:

```
var textField = UITextField()
```

In Swift, when you declare a variable of a particular type, it can also hold a reference to any subclass of that type. For example, take the class hierarchy shown in *Figure 2*, which shows **UITextField**, **UIButton**, and **UISlider**, just a few of the subclasses of the **UIControl** class.

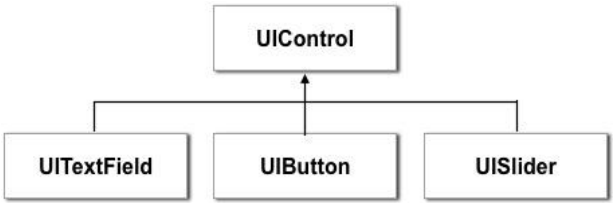

Figure 2 - When you declare a variable of a particular type, it can hold a Reference to any subclass of that type.

The word "polymorphism" means "many forms", and in this example you can see the **UIControl** class can take many different forms—a text field, a

2. Polymorphism[1]

Polymorphism is the provision of a single interface to entities of different types.
Here's a simple example in Ruby :

```
1 class Document
2   def initialize
3   end
4
5   def print
6     raise NotImplementedError, 'You must
7 implement the print method'
8   end
9 end
```

```
1  class XmlDocument < Document
2
3    def print
4      p 'Print from XmlDocument'
5    end
6
7  end
```

```
1  class HtmlDocument < Document
2
3    def print
4      p 'Print from HtmlDocument'
5    end
6  end
```

button, or a switch.

Given this hierarchy, you can declare a variable of type **UIControl** and then store a reference to the **UITextField**, **UIButton** or **UISwitch** object in this variable:

```
var control: UIControl
control = UITextField()
control = UIButton()
control = UISwitch()
```

```
1 XmDocument.new.print # Print from
2 XmlDocument
3 HtmlDocument.new.print # Print from
  HtmlDocument
```

As you can see, we sent the same message to different object and got different result. The print vmethod is a single interface to entities of different types: **XmlDocument** and **HtmlDocument.**

3. Encapsulation[4]

Encapsulation is one of the most important object-oriented design principles: It hides the internal states and functionality of objects. In Swift Encapsulation can be achieved by using the access control features of Swift.
The three access levels included in this release are:

- **private** entities are available only from within the source file where they are defined.
- **internal** entities are available to the entire module that includes the definition (e.g. an app or framework target).
- **public** entities are intended for use as API, and can be accessed by any file that imports the module, e.g. as a framework used in several of your projects.

In addition to allowing access specification for an entire declaration, Swift allows the get of a property to be more accessible than its set. Here is an example class that is part of a framework:

3. Encapsulation[5]

Ruby gives you three levels of protection:

1. **Public** methods can be called by everyone - no access control is enforced. *A class's instance methods (these do not belong only to one object; instead, every instance of the class can call them) are public by default*; anyone can call them. The **initialize** method is always private.
2. **Protected** methods can be invoked only by objects of the defining class and its subclasses. Access is kept within the family. However, usage of **protected** is limited.
3. **Private** methods cannot be called with an explicit receiver - the receiver is always **self**. This means that private methods can be called only in the context of the current object; you cannot invoke another object's private methods.

You can set access levels of named methods by listing them as arguments to the access control functions.

Encapsulation is achieved when the instance variables are private to an object and you have public getters and setters (in Ruby, we call them attribute readers and attribute writers). To make instance variables available, Ruby provides accessor methods that return their values.

```
public class ListItem {

	// Public properties.
	public var text: String
	public var isComplete: Bool

	// Readable throughout the
module, but only writeable from
within this file.
	private(set) var UUID:
NSUUID

	public init(text: String,
completed: Bool, UUID: NSUUID) {
		self.text = text
		self.isComplete =
completed
		self.UUID = UUID
	}

	// Usable within the
framework target, but not by
other targets.
	func refreshIdentity() {
		self.UUID = NSUUID()
	}

	public override func
isEqual(object: AnyObject?) ->
Bool {
		if let item = object
as? ListItem {
			return
self.UUID == item.UUID
		}
		return false
	}
}
```

The program p048accessor.rb illustrates the same.

```
# p048accessor.rb
# First without accessor methods
class Song
  def initialize(name, artist)
    @name     = name
    @artist   = artist
  end
  def name
    @name
.       end
.       def artist
.         @artist
.       end
.     end
.
.     song = Song.new("Brazil", "Ivete
Sangalo")
.     puts song.name
.     puts song.artist
.
.     # Now, with accessor methods
.     class Song
.       def initialize(name, artist)
.         @name     = name
.         @artist   = artist
.       end
.       attr_reader :name, :artist  # c
reate reader only
.       # For creating reader and write
r methods
.       # attr_accessor :name
.       # For creating writer methods
.       # attr_writer :name
.
.     end
.
.     song = Song.new("Brazil", "Ivete
Sangalo")
.     puts song.name
.     puts song.artist
```

Conclusion

Analyzing these two Object Oriented Programming languages Swift and Ruby, more specific the OOP concepts - inheritance, polymorphism, and encapsulation. I have concluded the following:

1. Swift and Ruby, both are single-inheritance languages, but in addition Ruby replicates a certain form of multi-inheritance through the use of modules as mix-ins.

2. Both in Swift and Ruby the Polymorphism means being able to send the same message to different objects and get different results

3. In Swift Encapsulation can be achieved by using the access control features of Swift (private, internal, public)

In Ruby, public, private and protected apply only to methods. Instance and class variables are encapsulated and effectively private, and constants are effectively public. There is no way to make an instance variable accessible from outside a class (except by defining an accessor method). And there is no way to define a constant that is inaccessible to outside use.

Bibliography

- http://samurails.com/interview/ruby-inheritance-encapsulation-polymorphism/ [1]
- http://www.raywenderlich.com/81952/intro-object-oriented-design-swift-part-1 [2]
- http://www.iphonelife.com/blog/31369/swift-programming-101-inheritance-polymorphism [3]
- https://developer.apple.com/swift/blog/?id=5 [4]
- http://rubylearning.com/satishtalim/ruby_access_control.html [5]