Ministry of Education of Republic of Moldova
Technical University of Moldova
CIM Faculty
Anglophone Department

**Report**

*on APPOO*

Laboratory Work Nr. 0

Performed by st. Gr. FAF-131 **Boldescu A.**

Verified by lect.univ **Drahnea V.**

Chișinău 2016

**Laboratory Work № 0**

**Topic:**

Warming up

**Task:**

- To select two languages, at least one of these to be Object Oriented (OO). To prove with examples (code) how we can implement core OOP concepts – **inheritance**, **polymorphism**, **encapsulation**.
- To make comparison analysis of this concepts in selected languages. Push your report to public repo and submit link to it.

**Short theory & methods used in this laboratory work:**

**Object-oriented programming (OOP)** is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A distinguishing feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of *"this"* or *"self").*

**Object**

This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

*Example* ***Dog named Steal***

**Class**

When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

*Example* ***Animal class for a Dog object.***

**Abstraction**

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

**Encapsulation**

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

**Inheritance**

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

**Polymorphism**

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

**Overloading**

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

**Solution:**

**Task 1:**

**Encapsulation**

**Ruby**

Ruby gives you three levels of protection at instance methods level which may be **public, private, or protected**. Ruby does not apply any access control over instance and class variables.

- **Public Methods:** Public methods can be called by anyone. Methods are public by default except for initialize, which is always private.
- **Private Methods:** Private methods cannot be accessed, or even viewed from outside the class. Only the class methods can access private members.
- **Protected Methods:** A protected method can be invoked only by objects of the defining class and its subclasses. Access is kept within the family.

Following is a simple example to show the syntax of all the three access modifiers:

```
#!/usr/bin/ruby -w

# define a class
class Box
  # constructor method
  def initialize(w,h)
    @width, @height = w, h
  end

  # instance method by default it is public
  def getArea
    getWidth() * getHeight
  end

  # define private accessor methods
```

```ruby
  def getWidth
    @width
  end
  def getHeight
    @height
  end
  # make them private
  private :getWidth, :getHeight

  # instance method to print area
  def printArea
    @area = getWidth() * getHeight
    puts "Big box area is : #@area"
  end
  # make it protected
  protected :printArea
end

# create an object
box = Box.new(10, 20)

# call instance methods
a = box.getArea()
puts "Area of the box is : #{a}"

# try to call protected or methods
box.printArea()
```

When the above code is executed, it produces the following result. Here, first method is called successfully but second method gave a problem.

```
Area of the box is : 200
test.rb:42: protected method `printArea' called for #
<Box:0xb7f11280 @height=20, @width=10> (NoMethodError)
```

In Ruby, public, private and protected apply only to methods. Instance and class variables are encapsulated and effectively private, and constants are effectively public.

**C#**

C# supports the following access specifiers:

- Public
- Private
- Protected
- Internal
- Protected internal

**Public Access Specifier**

Public access specifier allows a class to expose its member variables and member functions to other functions and objects. Any public member can be accessed from outside the class.

The following example illustrates this:

```
using System;
namespace RectangleApplication
{
  class Rectangle
  {
    //member variables
    public double length;
    public double width;

    public double GetArea()
    {
```

```
      return length * width;
    }
    public void Display()
    {
      Console.WriteLine("Length: {0}", length);
      Console.WriteLine("Width: {0}", width);
      Console.WriteLine("Area: {0}", GetArea());
    }
  }//end class Rectangle

  class ExecuteRectangle
  {
    static void Main(string[] args)
    {
      Rectangle r = new Rectangle();
      r.length = 4.5;
      r.width = 3.5;
      r.Display();
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
Length: 4.5
Width: 3.5
Area: 15.75
```

In the preceding example, the member variables length and width are declared**public**, so they can be accessed from the function Main() using an instance of the Rectangle class, named **r**.

The member function *Display()* and *GetArea()* can also access these variables directly without using any instance of the class.

The member functions *Display()* is also declared **public**, so it can also be accessed from *Main()* using an instance of the Rectangle class, named **r**.

**Private Access Specifier**

Private access specifier allows a class to hide its member variables and member functions from other functions and objects. Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.

The following example illustrates this:

```
using System;
namespace RectangleApplication
{
  class Rectangle
  {
    //member variables
    private double length;
    private double width;

    public void Acceptdetails()
    {
      Console.WriteLine("Enter Length: ");
      length = Convert.ToDouble(Console.ReadLine());
      Console.WriteLine("Enter Width: ");
      width = Convert.ToDouble(Console.ReadLine());
    }
    public double GetArea()
    {
      return length * width;
```

```
    }
    public void Display()
    {
      Console.WriteLine("Length: {0}", length);
      Console.WriteLine("Width: {0}", width);
      Console.WriteLine("Area: {0}", GetArea());
    }
  }//end class Rectangle

  class ExecuteRectangle
  {
    static void Main(string[] args)
    {
      Rectangle r = new Rectangle();
      r.Acceptdetails();
      r.Display();
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
Enter Length:
4.4
Enter Width:
3.3
Length: 4.4
Width: 3.3
Area: 14.52
```

**Protected Access Specifier**

Protected access specifier allows a child class to access the member variables and member functions of its base class. This way it helps in implementing inheritance. We will discuss this in more details in the inheritance chapter.

**Internal Access Specifier**

Internal access specifier allows a class to expose its member variables and member functions to other functions and objects in the current assembly. In other words, any member with internal access specifier can be accessed from any class or method defined within the application in which the member is defined.

The following program illustrates this:

```
using System;
namespace RectangleApplication
{
  class Rectangle
  {
    //member variables
    internal double length;
    internal double width;

    double GetArea()
    {
      return length * width;
    }
    public void Display()
    {
      Console.WriteLine("Length: {0}", length);
      Console.WriteLine("Width: {0}", width);
      Console.WriteLine("Area: {0}", GetArea());
    }
```

```
  }//end class Rectangle


  class ExecuteRectangle
  {
    static void Main(string[] args)
    {
      Rectangle r = new Rectangle();
      r.length = 4.5;
      r.width = 3.5;
      r.Display();
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
Length: 4.5
Width: 3.5
Area: 15.75
```

In the preceding example, notice that the member function *GetArea()* is not declared with any access specifier. Then what would be the default access specifier of a class member if we don't mention any? It is **private**.

**Protected Internal Access Specifier**

The protected internal access specifier allows a class to hide its member variables and member functions from other class objects and functions, except a child class within the same application. This is also used while implementing inheritance.

**Polymorphism**

**Ruby**

```
class GenericParser
  def parse
    raise NotImplementedError, 'You must implement the parse method'
  end
end
class JsonParser < GenericParser
  def parse
    puts 'An instance of the JsonParser class received the parse message'
  end
end
class XmlParser < GenericParser
  def parse
    puts 'An instance of the XmlParser class received the parse message'
  end
end
puts 'Using the XmlParser'
parser = XmlParser.new
parser.parse

puts 'Using the JsonParser'
parser = JsonParser.new
parser.parse
```

The parse method is a single interface to entities of different types : XmlParser and JsonParser.

**C#**

In C# Polymorphism can be static or dynamic. In **static polymorphism**, the response to a function is determined at the compile time. In **dynamic polymorphism**, it is decided at run-time.

**Static Polymorphism**

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are:

- Function overloading
- Operator overloading

**Function Overloading**

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

The following example shows using function **print()** to print different data types:

```
using System;
namespace PolymorphismApplication
{
  class Printdata
  {
    void print(int i)
    {
      Console.WriteLine("Printing int: {0}", i );
    }

    void print(double f)
    {
      Console.WriteLine("Printing float: {0}" , f);
    }

    void print(string s)
    {
      Console.WriteLine("Printing string: {0}", s);
    }
    static void Main(string[] args)
    {
      Printdata p = new Printdata();

      // Call print to print integer
```

```
        p.print(5);

        // Call print to print float
        p.print(500.263);

        // Call print to print string
        p.print("Hello C++");
        Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Printing int: 5
Printing float: 500.263
Printing string: Hello C++
```

**Dynamic Polymorphism**

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. **Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

Here are the rules about abstract classes:

- You cannot create an instance of an abstract class
- You cannot declare an abstract method outside an abstract class
- When a class is declared **sealed**, it cannot be inherited, abstract classes cannot be declared sealed.

The following program demonstrates an abstract class:

```
using System;
namespace PolymorphismApplication
{
  abstract class Shape
  {
    public abstract int area();
  }
  class Rectangle:  Shape
  {
    private int length;
    private int width;
    public Rectangle( int a=0, int b=0)
    {
      length = a;
      width = b;
    }
    public override int area ()
    {
      Console.WriteLine("Rectangle class area :");
      return (width * length);
    }
  }

  class RectangleTester
  {
    static void Main(string[] args)
```

```
    {
      Rectangle r = new Rectangle(10, 7);
      double a = r.area();
      Console.WriteLine("Area: {0}",a);
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
Rectangle class area :
Area: 70
```

When you have a function defined in a class that you want to be implemented in an inherited class(es), you use **virtual** functions. The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

The following program demonstrates this:

```
using System;
namespace PolymorphismApplication
{
  class Shape
  {
    protected int width, height;
    public Shape( int a=0, int b=0)
    {
      width = a;
      height = b;
    }
    public virtual int area()
```

```
  {
    Console.WriteLine("Parent class area :");
    return 0;
  }
}
class Rectangle: Shape
{
  public Rectangle( int a=0, int b=0): base(a, b)
  {

  }
  public override int area ()
  {
    Console.WriteLine("Rectangle class area :");
    return (width * height);
  }
}
class Triangle: Shape
{
  public Triangle(int a = 0, int b = 0): base(a, b)
  {

  }
  public override int area()
  {
    Console.WriteLine("Triangle class area :");
    return (width * height / 2);
  }
}
```

```
  class Caller
  {
    public void CallArea(Shape sh)
    {
      int a;
      a = sh.area();
      Console.WriteLine("Area: {0}", a);
    }
  }
  class Tester
  {

    static void Main(string[] args)
    {
      Caller c = new Caller();
      Rectangle r = new Rectangle(10, 7);
      Triangle t = new Triangle(10, 5);
      c.CallArea(r);
      c.CallArea(t);
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
Rectangle class area:
Area: 70
Triangle class area:
Area: 25
```

**Inheritance**

**Ruby**

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.

A class can only inherit from one class as opposed as well as in C# where multi- class inheritance can't be done.

You can however replicate a certain form of multi-inheritance through the use of modules as mix-ins:

```
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
```

**C#**

In C# A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base classes or interfaces.

The syntax used in C# for creating derived classes is as follows:

```
<acess-specifier> class <base_class>
{
   ...
}
class <derived_class> : <base_class>
```

```
{
   ...
}
```

Consider a base class Shape and its derived class Rectangle:

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      public void setHeight(int h)
      {
         height = h;
      }
      protected int width;
      protected int height;
   }

   // Derived class
   class Rectangle: Shape
   {
      public int getArea()
      {
         return (width * height);
      }
   }

   class RectangleTester
   {
```

```
      static void Main(string[] args)
      {
         Rectangle Rect = new Rectangle();

         Rect.setWidth(5);
         Rect.setHeight(7);

         // Print the area of the object.
         Console.WriteLine("Total area: {0}",  Rect.getArea());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Total area: 35
```

**Initializing Base Class**

The derived class inherits the base class member variables and member methods. Therefore the super class object should be created before the subclass is created. You can give instructions for superclass initialization in the member initialization list.

The following program demonstrates this:

```
using System;
namespace RectangleApplication
{
   class Rectangle
   {
      //member variables
      protected double length;
```

```
    protected double width;
    public Rectangle(double l, double w)
    {
       length = l;
       width = w;
    }

    public double GetArea()
    {
       return length * width;
    }

    public void Display()
    {
       Console.WriteLine("Length: {0}", length);
       Console.WriteLine("Width: {0}", width);
       Console.WriteLine("Area: {0}", GetArea());
    }
}//end class Rectangle

class Tabletop : Rectangle
{
    private double cost;
    public Tabletop(double l, double w) : base(l, w)
    { }
    public double GetCost()
    {
       double cost;
       cost = GetArea() * 70;
       return cost;
    }
    public void Display()
    {
```

```
         base.Display();
         Console.WriteLine("Cost: {0}", GetCost());
      }
   }
   class ExecuteRectangle
   {
      static void Main(string[] args)
      {
         Tabletop t = new Tabletop(4.5, 7.5);
         t.Display();
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Length: 4.5
Width: 7.5
Area: 33.75
Cost: 2362.5
```

## Multiple Inheritance in C#

**C# does not support multiple inheritance**. However, you can use interfaces to implement multiple inheritance. The following program demonstrates this:

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
```

```
   {
      width = w;
   }
   public void setHeight(int h)
   {
      height = h;
   }
   protected int width;
   protected int height;
}

// Base class PaintCost
public interface PaintCost
{
   int getCost(int area);

}
// Derived class
class Rectangle : Shape, PaintCost
{
   public int getArea()
   {
      return (width * height);
   }
   public int getCost(int area)
   {
      return area * 70;
   }
}
class RectangleTester
{
   static void Main(string[] args)
   {
```

```
         Rectangle Rect = new Rectangle();
         int area;
         Rect.setWidth(5);
         Rect.setHeight(7);
         area = Rect.getArea();

         // Print the area of the object.
         Console.WriteLine("Total area: {0}",  Rect.getArea());
         Console.WriteLine("Total paint cost: ${0}" , Rect.getCost(area));
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Total area: 35
Total paint cost: $2450
```

**Task 2:**

***Encapsulation***

**C#**

- Public, protected, private access modifiers applicable to fields, methods, classes.

**Ruby**

- Public, protected, private access modifiers applicable only methods.

***Inheritance***

- Child class can inherit from only one parent class for both languages.

**C#**

- Multiple interface inheritance

**Ruby**

- Usage of mixins

**Polymorphism**

**C#**

- Polymorphism and inheritance are linked

**Ruby**

- Full separation of polymorphism and inheritance

**Conclusions:**

After this laboratory work I learned some basic features of Ruby language especially regarding OOP principles also I tried to figure out some differences between C# and ruby regarding OOP .