Topic: Interfaces.
OOP concepts involved: Inheritance, Abstract Classes, Class Instantiation and Declaration, Class Members, Polymorphism, Constructor Methods, Class Modifiers, Interfaces, Getters and Setters.
Programming generic concepts involved: Access Modifiers, Functions, Primitive Variables, Variable Declaration, Function Parameters.

➢ Theoric introduction

Interfaces

In Java, an interface is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods and nested types. Method bodies exist only for default and static methods.

An important factor is that Interfaces cannot be instantiated, they can only be implemented by classes or extended by other interfaces. Unless a class that implement an interface is abstract, all the methods of the interface must be defined in the class.

Declaring an interface is very similar to declaring a class:

```
[Access Modifier] interface [identifier]{ }
```

An interface may seem similar to an abstract class, but there are key differences between the two.

- ❏ All methods in an interface must be declared as abstract and public
- ❏ An interface cannot implement any method, all methods must be abstract
- ❏ An interface cannot be instanced
- ❏ A class can implement multiple interfaces, but it can only extend one class
- ❏ Classes are part of a hierarchy, while interfaces are not. Multiple classes can implement the same interface with no connection between them.

Implementing an Interface

After we have created our interface, we now need a way to implement it into a class. To do this we use the reserved keyword **implements** in the class header.

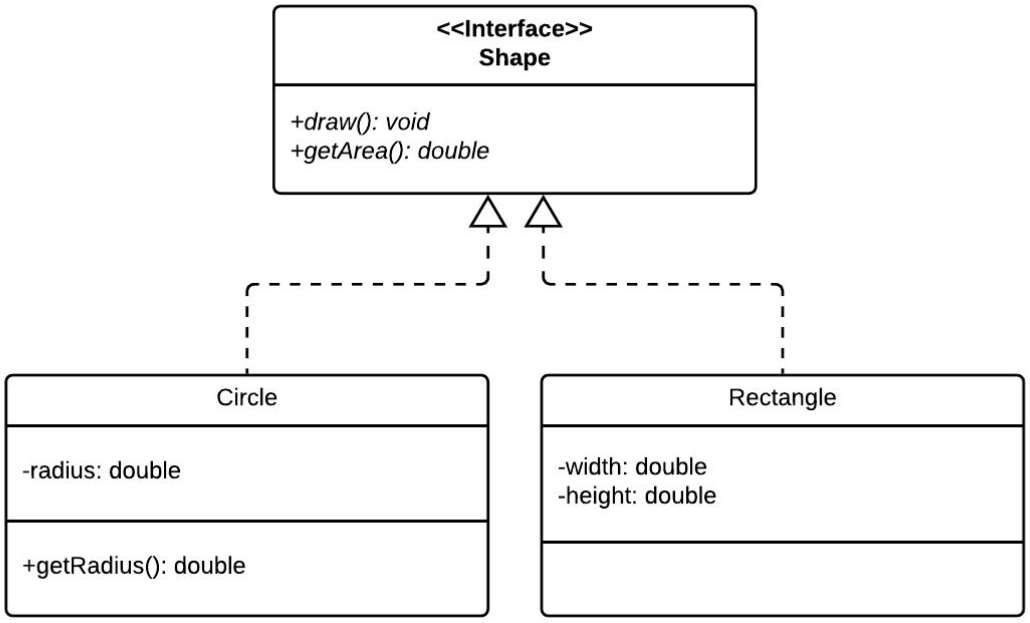
```
[Access Modifier] [Modifiers] class [identifier] implements [interface identifier]{ }
```

When implement an interface, we must declare all abstract methods inside the interface.

➢ Statement

Develop a program that creates an interface where the basic functions of a geometric shape are. Create at least 2 shape classes that implement that interface to test its functionality.

➢ Class design (UML)



➢ Program Code

Shape.java

```
public interface Shape {
	//Declaring the interface abstract methods
	void draw();
	double getArea();
}
```

Circle.java

```
public class Circle implements Shape {

	private double radius;

	public Circle(double r){
		this.radius = r;
	}
	// Making our own implementation of the parent abstract methods
	@Override
	public void draw() {
		System.out.println("Drawing Circle");
	}

	@Override
	public double getArea(){
		return Math.PI*this.radius*this.radius;
	}

	public double getRadius(){
		return this.radius;
	}
}
```

Rectangle.java

```
public class Rectangle implements Shape {

	private double width;
	private double height;

	public Rectangle(double w, double h){
		this.width=w;
```

```
                this.height=h;
        }
        // Making our own implementation of the parent abstract methods
        @Override
        public void draw() {
                System.out.println("Drawing Rectangle");
        }

        @Override
        public double getArea() {
                return this.height*this.width;
        }

}
```

Main.java

```
public class Main{

        public static void main(String[] args) {

                //programming for interfaces not implementation
                Shape shape = new Circle(10);

                shape.draw();
                System.out.println("Area="+shape.getArea());

                //switching from one implementation to another easily
                shape=new Rectangle(10,10);
                shape.draw();
                System.out.println("Area="+shape.getArea());
        }
}
```

➢ Program execution

Implementing the interface “Shape” allowed us to create a template for all Shape Objects we create in the future. In this case Circle and Rectangle both must follow the same template. After we create the objects and call their methods, we should see the area of both of them.

```
Drawing Circle
Area=314.1592653589793
Drawing Rectangle
Area=100.0
```

➢ Conclusions

Interfaces are a great tool in projects where multiple programmers are working at the same time. In this cases, interfaces work as kind of “contracts” that spells out how their software interacts. This way, all programmers should be able to write their code without knowledge of how other programmers wrote their code.