Java 1

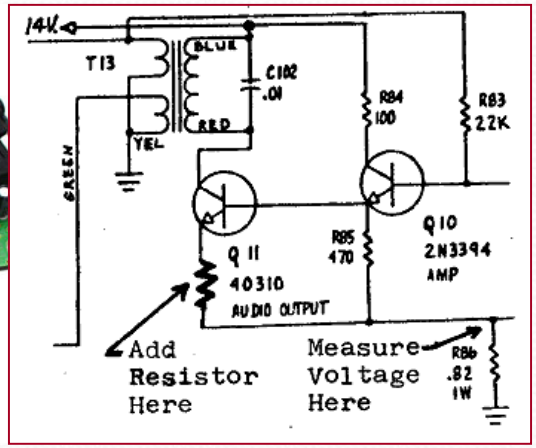
Lecture 07

Sayed Ahmad Sahim

18.October.2017

Encapsulation

- **encapsulation**: Hiding implementation details of an object from its clients.
  - Encapsulation provides *abstraction*.
    - separates external view (behavior) from internal view (state)
  - Encapsulation protects the integrity of an object's data.



Private fields

- A field can be declared *private*.
  - No code outside the class can access or change it.

```
private type name;
```

  - Examples:

```
private int id;
private String name;
```

- Client code sees an error when accessing private fields:

```
PointMain.java:11: x has private access in Point
System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
                                  ^
```

Accessing private state

- We can provide methods to get and/or set a field's value:

```
// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}

// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}
```

  - Client code will look more like this:

```
System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
p1.setX(14);
```

Point class, version 4

```
// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }

    public void translate(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
```

Client code, version 4

```
public class PointMain4 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);

        // print each point
        System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");

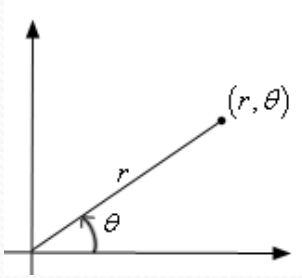
        // move p2 and then print it again
        p2.translate(2, 4);
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");
  }
}
```

OUTPUT:

```
p1 is (5, 2)
p2 is (4, 3)
p2 is (6, 7)
```

Benefits of encapsulation

- Provides abstraction between an object and its clients.
- Protects an object from unwanted access by clients.
  - A bank app forbids a client to change an `Account`'s balance.
- Allows you to change the class implementation.
  - `Point` could be rewritten to use polar coordinates (radius *r*, angle *θ*), but with the same methods.



- Allows you to constrain objects' state (**invariants**).
  - Example: Only allow `Points` with non-negative coordinates.

Java Abstraction

Abstraction in Java

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- Abstraction lets you focus on what the object does instead of how it does it.
- Ways to achieve Abstraction
  - Abstract class
  - Interface

Abstract class in Java

- A class that is declared as abstract is known as abstract class.
- The class needs to be extended and its methods are implemented by the child class.
- Abstract class cannot be instantiated.
- Syntax:
  - abstract class A{}

abstract method

- A method that is declared as abstract and does not have implementation is known as abstract method.
- Example abstract method
  - ```
    abstract void printStatus();//no body and abstract
    ```
    ```

Example

- In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely..");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

Example

```
abstract class Bank{
    abstract int getRateOfInterest();
}
class SBI extends Bank{
    int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
    int getRateOfInterest(){return 8;}
}

class TestAbstraction{
    public static void main(String args[]){
        Bank b;
        b=new SBI();
        System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
        b=new PNB();
        System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");

    }

}
```

Abstract Class Members

- Abstract method
- Methods
- data member
- Constructor
- main() method

```
abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
 }

 class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }
 class TestAbstraction2{
 public static void main(String args[]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }
}
```

Abstract class and interface

- The abstract class can also be used to provide some implementation of the interface.
- In such case, the end user may not be forced to override all the methods of the interface.

```
interface A{
    void a();
    void b();
    void c();
    void d();
}

abstract class B implements A{
    public void c(){System.out.println("I am C");}
}

class M extends B{
    public void a(){System.out.println("I am a");}
    public void b(){System.out.println("I am b");}
    public void d(){System.out.println("I am d");}
}

class Test5{
    public static void main(String args[]){
    A a=new M();
    a.a();

    }
}
```

Interface in Java

Interface in Java

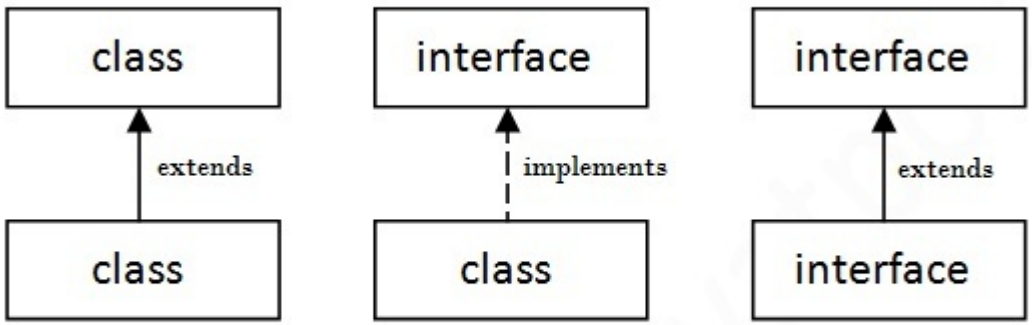
- An **interface** in java is a blueprint of a class.
- It has static constants and abstract methods.
- The interface in java is a mechanism to achieve abstraction.
- There can be only abstract methods in the java interface not method body.
- Java Interface also represents **IS-A relationship**.
- It cannot be instantiated just like abstract class.

Why use Java interface?

- There are mainly three reasons to use interface.
  - It is used to achieve abstraction.
  - By using interface, we can support the functionality of multiple inheritance.
  - It can be used to achieve loose coupling.

Classes vs Interfaces

- A class extends another class.
- An interface extends another interface but a class implements an interface.



Example

```
interface printable{
    void print();
}
class A6 implements printable{
    public void print(){System.out.println("Hello");}

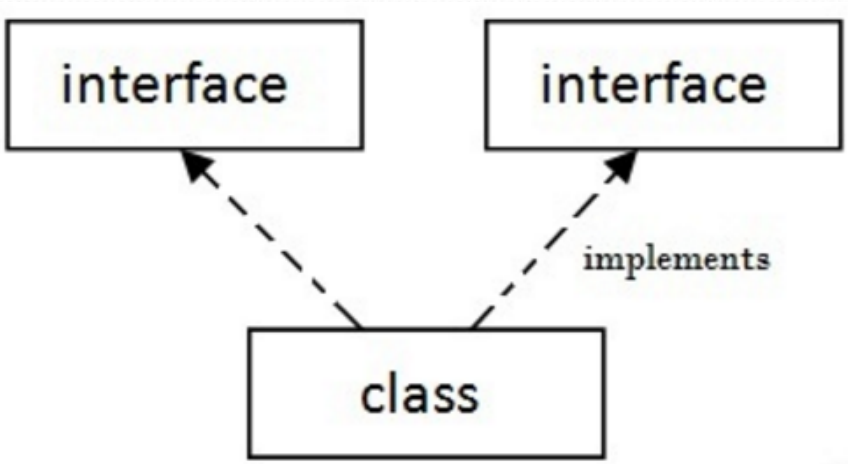
    public static void main(String args[]){
        A6 obj = new A6();
        obj.print();
    }
}
```

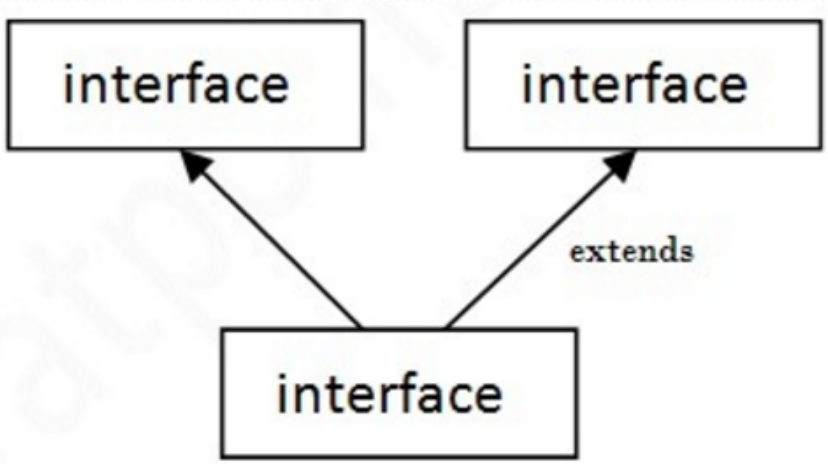
Example II

```
interface Drawable{
    void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
    public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
    public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
    public static void main(String args[]){
        Drawable d=new Circle();//In real scenario, object is provided
        d.draw();
    }
}
```

Multiple inheritance

- If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.





Example

```
interface Printable{
    void print();
}
interface Showable{
    void show();
}
class A7 implements Printable,Showable{
    public void print(){System.out.println("Hello");}
    public void show(){System.out.println("Welcome");}

public static void main(String args[]){
    A7 obj = new A7();
    obj.print();
    obj.show();
    }
}
```

Interface inheritance

- A class implements interface but one interface extends another interface .

```
interface Printable{
    void print();
}
interface Showable extends Printable{
    void show();
}
class TestInterface4 implements Showable{
    public void print(){System.out.println("Hello");}
    public void show(){System.out.println("Welcome");}

public static void main(String args[]){
    TestInterface4 obj = new TestInterface4();
    obj.print();
    obj.show();
 }
}
```

Java 8 Default Method

- Since Java 8, we can have method body in interface. But we need to make it default method.

```
interface Drawable{
    void draw();
    default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
    public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
    public static void main(String args[]){
        Drawable d=new Rectangle();
        d.draw();
        d.msg();
}}
```

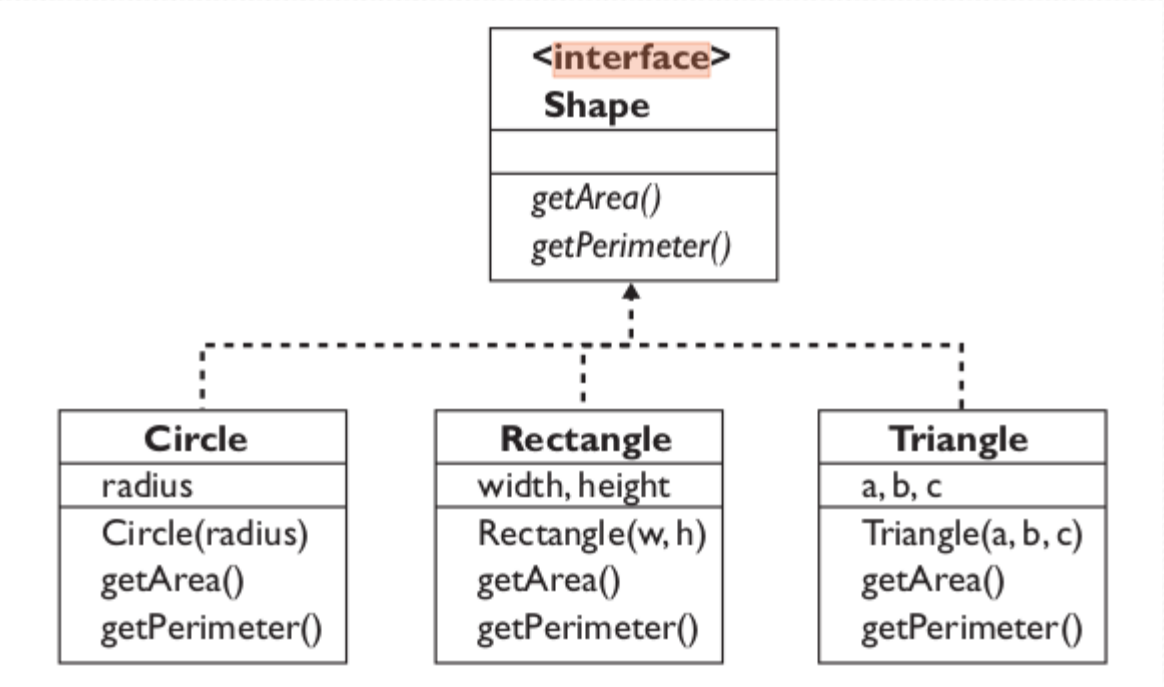
Marker Interface

- An interface that have no member is known as marker or tagged interface.
- For example: Serializable, Cloneable, Remote etc.
- They are used to provide some essential information to the JVM so that JVM may perform some useful operation.
-

```
public interface Serializable{

}
```

Exercise


<interface>
Shape
getArea()
getPerimeter()
Circle
radius
Circle(radius)
getArea()
getPerimeter()
Rectangle
width, height
Rectangle(w, h)
getArea()
getPerimeter()
Triangle
a, b, c
Triangle(a, b, c)
getArea()
getPerimeter()