National University of Computer & Emerging Sciences,
Karachi
Computer Science Department
Spring 2024, Lab Manual – 08

Course Code: CL-217	**Course : Object Oriented Programming Lab**
Instructor(s) :	**Ali Fatmi**

LAB - 8

Abstract Classes & Interface

Abstract Class:

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Syntax

abstract *class MyClass*
{ // code
}

Abstract Method:

A method which is declared as abstract and does not have implementation is known as an **abstract method**.

Syntax

abstract *void myFunction( ); //no method body and abstract*

Example

```
abstract public class Employee {
    abstract public void work();
    // abstract function

}
```

It is the responsibility of child class(es) to override the abstract function and “complete” the parent class.

Example

```
public class Pilot extends Employee {
    public void work()
    {
        System.out.println("Flying the plane");
    }
}
```

Example: (*Using abstract class from main)*

```
public class MainClass {
    public static void main(String args[]) {
        Employee e = new Pilot();
        e.work();
    }
}
```

Output:

```
Flying the plane

Process finished with exit code 0
```

Point to Remember: We cannot create an instance of the abstract class.
Point to Remember: An abstract class can contain concrete functions.

Example:

```
abstract public class Employee {
    abstract public void work();
    // abstract function
    void sign_in() {
        System.out.println("Employee signing in...");
    }
}
```

```
public class MainClass {
    public static void main(String args[]) {
        Employee e = new Pilot();
        e.sign_in();
    }
}
```

Anonymous Class:

If we do not have a child class, we can still override the abstract function by creating an **Anonymous Class.**

- A new class is defined (without a name, so called anonymous class)
- This new class extends abstract base class
- Abstract methods are overriden in this new class
- New instance of this new class is created and assigned to the parent variable

Example

```
public class MainClass {
    public static void main(String args[]) {
        Employee e = new Employee() {
            @Override
            public void work() {
                System.out.println("Employee is working");
            }
        };
    }
}
```

Interfaces:

An **interface** in Java is a blueprint of a class. It has static constants and abstract methods. A difference between abstract classes and interface is that there can be concrete methods in abstract classes whereas interface cannot contain any.

How to create an interface:

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

interface *<interface_name>{*

```
    // declare constant fields
    // declare methods that are abstract
    // by default.
}
```

Example: *(Declaring an interface)*

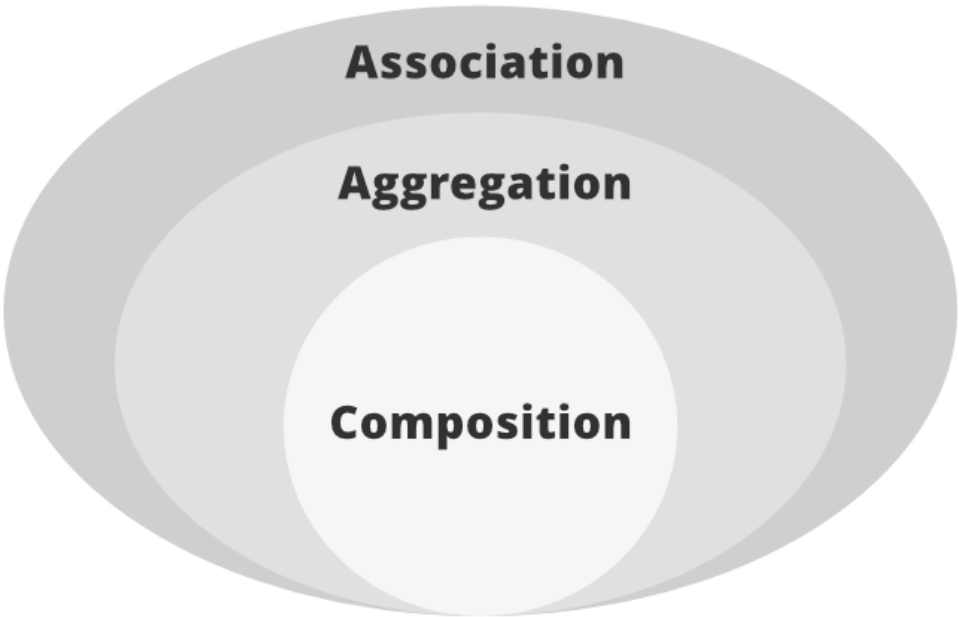
```java
public interface Printable {
    void printMsg();
}
```

Example: *(Implementing interface in class)*

```java
public class PrinterClass implements Printable {
    public void printMsg()
    {
        System.out.println("This is my implementation");
    }
}
```

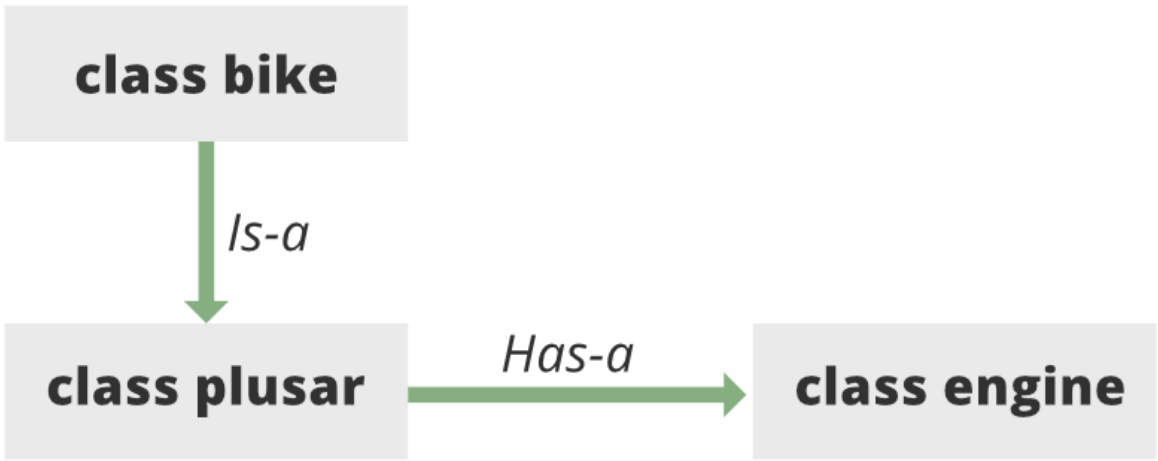
Association

Association is the relation between two separate classes which establishes through their Objects. Composition and Aggregation are the two forms of association. In Java, a Has-A relationship is otherwise called composition. It is additionally utilized for code reusability in Java. In Java, a Has-A relationship essentially implies that an example of one class has a reference to an occasion of another class or another occurrence of a similar class. For instance, a vehicle has a motor, a canine has a tail, etc. In Java, there is no such watchword that executes a Has-A relationship. Yet, we generally utilize new catchphrases to actualize a Has-A relationship in Java.



- It represents the Has-A relationship.
- It is a unidirectional association i.e. a one-way relationship. For example, here above as shown pulsar motorcycle has an engine but vice-versa is not possible and thus unidirectional in nature.
- In Aggregation, both the entries can survive individually which means ending one entity will not affect the other entity.

.Illustration:



This shows that class Pulsar Has-a an engine. By having a different class for the engine, we don't need to put the whole code that has a place with speed inside the Van class, which makes it conceivable to reuse the Speed class in numerous applications.

In an Object-Oriented element, the clients don't have to make a big deal about which article is accomplishing the genuine work. To accomplish this, the Van class conceals the execution subtleties from the clients of the Van class. Thus, essentially what happens is the clients would ask the Van class to do a specific activity and the Van class will either accomplish the work without help from anyone else or request that another class play out the activity.

Implementation: Here is the implementation of the same which is as follows:

- Car class has a couple of instance variable and few methods
- Maserati is a type of car that extends the Car class that shows Maserati is a Car. Maserati also uses an Engine's method, stop, using composition. So it shows that a Maserati has an Engine.
- The Engine class has the two methods start() and stop() that are used by the Maserati class.

```
// Java Program to Illustrate has-a relation

// Class1
// Parent class
public class Car {

    // Instance members of class Car
    private String color;
    private int maxSpeed;

    // Main driver method
    public static void main(String[] args)
    {
        // Creating an object of Car class
```

```
        Car nano = new Car();

        // Assigning car object color
        nano.setColor("RED");

        // Assigning car object speed
        nano.setMaxSpeed(329);

        // Calling carInfo() over object of Car class
        nano.carInfo();

        // Creating an object of Maserati class
        Maserati quattroporte = new Maserati();

        // Calling MaseratiStartDemo() over
        // object of Maserati class
        quattroporte.MaseratiStartDemo();
    }

    // Methods implementation

    // Method 1
    // To set the maximum speed of car
    public void setMaxSpeed(int maxSpeed)
    {
        // This keyword refers to current object itself
        this.maxSpeed = maxSpeed;
    }

    // Method 2
    // To set the color of car
    public void setColor(String color)
    {
        // This keyword refers to current object
        this.color = color;
    }

    // Method 3
    // To display car information
    public void carInfo()
    {
        // Print the car information - color and speed
        System.out.println("Car Color= " + color
                           + " Max Speed= " + maxSpeed);
    }
}

// Class2
```

```java
// Child class
// Helper class
class Maserati extends Car {

    // Method in which it is shown
    // what happened with the engine of Puslar
    public void MaseratiStartDemo()
    {
        // Creating an object of Engine type
        // using stop() method
        // Here, MaseratiEngine is name of an object
        Engine MaseratiEngine = new Engine();
        MaseratiEngine.start();
        MaseratiEngine.stop();
    }
}

// Class 3
// Helper class
class Engine {

    // Method 1
    // To start a engine
    public void start()
    {
        // Print statement when engine starts
        System.out.println("Started:");
    }

    // Method 2
    // To stop a engine
    public void stop()
    {
        // Print statement when engine stops
        System.out.println("Stopped:");
    }
}
```

Output

```
Car Color= RED Max Speed= 150

Started:

Stopped:
```

Lab Tasks:

1. Create an abstract class 'Parent' with a method 'message'. It has two subclasses each having a method with the same name 'message' that prints "This is first subclass" and "This is second subclass" respectively. Call the methods 'message' by creating an object for each subclass.

2. Create an abstract class 'Bank' with an abstract method 'getBalance'. $100, $150 and $200 are deposited in banks A, B and C respectively. 'BankA', 'BankB' and 'BankC' are subclasses of class 'Bank', each having a method named 'getBalance'. Call this method by creating an object of each of the three classes.

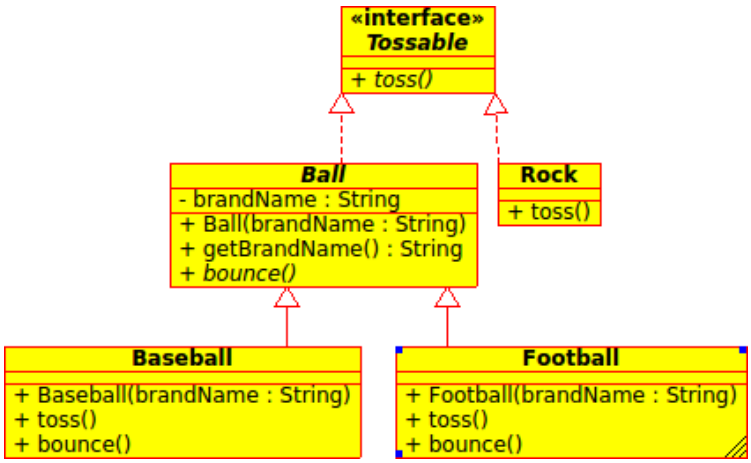
3. We have to calculate the percentage of marks obtained in three subjects (each out of 100) by student A and in four subjects (each out of 100) by student B. Create an abstract class 'Marks' with an abstract method 'getPercentage'. It is inherited by two other classes 'A' and 'B' each having a method with the same name which returns the percentage of the students. The constructor of student A takes the marks in three subjects as its parameters and the marks in four subjects as its parameters for student B. Create an object for eac of the two classes and print the percentage of marks for both the students.

4. An abstract class has a construtor which prints "This is constructor of abstract class", an abstract method named 'a_method' and a non-abstract method which prints "This is a normal method of abstract class". A class 'SubClass' inherits the abstract class and has a method named 'a_method' which prints "This is abstract method". Now create an object of 'SubClass' and call the abstract method and the non-abstract method. (Analyse the result)

5. Create an abstract class 'Animals' with two abstract methods 'cats' and 'dogs'. Now create a class 'Cats' with a method 'cats' which prints "Cats meow" and a class 'Dogs'

with a method 'dogs' which prints "Dogs bark", both inheriting the class 'Animals'. Now create an object for each of the subclasses and call their respective methods.

6. Implement the following class hierarchy on paper. You do not need to fill in the method bodies for the *toss* or *bounce* methods.



7. A queue is an abstract data type for adding and removing elements. The first element added to a queue ist the first element that is removed (first-in-first-out, FIFO). Queues can be used, for instance, to manage processes of an operating system: the first process added to the waiting queue is reactivated prior to all other processes (with the same priority).

   Design an interface Queue, with methods to add and remove elements (integers). Furthermore, a methode to check whether the queue is empty or not should exist.

   - Good luck!