Regarding **Abstraction**

Abstraction is the property with which only the essential details are displayed to the user.

The internal details or the non-essentials details are not displayed to the user.

(It means hiding complexities or hiding the implementation details from end user)

Real life example -

Consider a person walking to an ATM . She only knows how to withdraw / deposit money. But as the end user , she does not really need to know about how ATM connects with the underlying bank to inform about this transaction or any of the hardware used inside of the ATM.

This is what abstraction is.

In Java , **abstraction is already achieved using unit of encapsulation : class**

When you define methods(functionality) in the class , it's user(Client code using these methods) does not need to know about the actual implementation of the methods, it just needs to know about the invocation.

It can be achieved even further by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

When to use abstract classes and abstract methods ?

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.(i.e. only provide declaration)

Sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

eg : BoundedShape & it's method area

Here BoundedShape class does not know how to compute the area since it’s an abstract shape.

abstract is a keyword in Java

- Appears at the method and class level.

Abstract methods

- Methods only with declaration & no definition.

eg : public abstract double computeArea();

Can you declare the methods with -

private & abstract

final & abstract

static & abstract

- NO , it will result into javac error (since abstract methods must be overridden by the subclasses and private | final | static methods can't be overridden)

Rules

1. The class containing single or multiple abstract methods , has to be declared abstract.

eg : public abstract class Shape { .....

//abstract method

public abstract double computeArea();

}

2. You cannot create instance of the abstract class BUT can create abstract super class reference referring to a concrete (non abstract) sub class instance.

eg :

Shape shape=new Shape(...);//java compiler error - on RHS

Shape shape=new Circle(...);//works !

3. Can abstract class contain a parameterized constructor ?

- Yes . (It will be used for initializing the state of the concrete sub class object)

4. Can you declare an abstract class , without ANY of the abstract methods ?

- Yes

eg - Event Adapter classes or HttpServlet

5. Abstract classes can contain concrete methods

- It represents partial abstraction.

6. Can a class be declared as abstract & final ?

- NO

7. Can you create abstract subclass from the concrete super class ?

- Yes

Regarding "**final" keyword in java**

- It represents a constant in Java.

Usages

1 final primitive data member

- A constant (immutable)
- read only

eg -- public final int DATA=123;

DATA++;//javac error

2. **final methods** cannot be overridden.

eg -- Object class methods

- wait , notify ,notifyAll

3. **final class** can't be sub-classed(or extended)

eg -- String ,StringBuffer,StringBuilder

4. **final reference**

- Can't be re-assigned.

Eg. final Emp e=new Mgr(…);

e=new Worker(.....);//compiler err

Regarding Java Interfaces

What is an interface ?

- An interface in java is a blueprint of a class.
- Typically it has public static final data members
- public and abstract methods.
- It is a mechanism to achieve **complete abstraction**.
- Till JDK 1.7, one could only add abstract methods in the java and not its implementation.
- Java Interface also represents **IS-A relationship** , with the implementation class

- It cannot be instantiated just like abstract class.

Why java interfaces?

1. It is used to achieve **complete abstraction**.

2. Using interface, we can support the functionality of **multiple inheritance**(since a class can implement multiple interfaces)

3. It can be used to achieve **loose coupling**.

- Interfaces allow complete separation between WHAT(specification or a contract) is to be done Vs HOW (implementation details) it's to be done.

eg : In Collection Framework List interface specifies the behaviour to add or remove or search an element. But actual implementation details are left to its concrete implementation classes namely ArrayList , LinkedList or Vector

Features

The java compiler implicitly adds

- public and abstract keywords before the interface method
- public, static and final keywords before data members.

Syntax of interface

default(no modifier)/public interface NameOfInterface extends comma separated list of super interfaces {

```
//data members : implicitly public static final

//methods :    public abstract implicitly
```

}

Implementation class syntax

default(no modifier)/public class NameOfClass **extends** SuperCls **implements** comma separated list of interfaces {

- An implementation class **must** define| implement , all abstract methods inherited from all of these interfaces. If not then it must be declared as an abstract class.
- It can add new methods also.

}

eg : public class Circle extends BoundedShape implements AreaComputable, PerimeterComputable {...}

The optional but recommended annotation to tell java compiler about implementing methods

- @Override

1. **Relationship between classes and interfaces**

A class inherits from another class(extends), an interface extends another interfaces(extends) but a class implements an interface.

2. Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

eg :

```
interface Printable{    void print();    }

interface Computable{ double compute(double a,double b)}

class A implements Printable, Computable {

@Override public void print(){System.out.println("Hello");}

@Override public double compute(double a,double b){ return a+b;}

public static void main(String args[]){

A ref = new A();

ref.print();

ref.compute(10,20);

 }

}
```

Questions

Multiple inheritance is not supported through class in java but it is possible by interface, why?

Multiple inheritance is not supported in case of class, since it can create an ambiguity(diamond issue). But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class.

For example:

```
interface Printable{

void print();
```

```
}

interface Showable{

void print();

}

class TestTnterface1 implements Printable,Showable{

public void print(){System.out.println("Hello");}

public static void main(String args[]){

TestTnterface1 obj = new TestTnterface1();

obj.print();

  }

}
```

As you can see in the above example, Printable and Showable interface have same methods but its single implementation is provided by class TestTnterface1, so there is no ambiguity.

2. What about Interface inheritance ?

A class implements multiple interfaces but one interface extends another interfaces .

3. What **is marker or tag interface**?

An interface that has no member is known as marker or tag interface.

Eg. Serializable, Cloneable, Remote , SingleThreadModel

They are used to provide some essential information to the JVM(Run time marker) so that JVM may perform some useful operation.

4. **You can nest (i.e declare within interface)**

- Interfaces (implicitly public static)
- Classes (implicitly public static)
- Enums (implicitly public static)
- Annotations (typically used for marker annotations)
- default methods (Java 8 onwards)
- static methods (Java 8 onwards)

- private methods (Java 9 onwards)

5. **What is a functional interface ?**

- It is an interface containing exactly one abstract method (Single Abstract Method - SAM)

- It can optionally additionally contain other default , static , private methods

- Optional annotation

  - @FunctionalInterface (meant for Java compiler)

eg : Comparator , Runnable , Consumer, Predicate