**ABSTRACT:** Sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement.

You may have methods that must be overridden by the subclass in order for the subclass to have any meaning.

In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java’s solution to this problem is the abstract method.

You can require that certain methods be overridden by subclasses by specifying the abstract type modifier.

abstract type name(parameter-list);

These methods are sometimes referred to as subclass's responsibility because they have no implementation specified in the superclass.

Thus, a subclass must override them—it cannot simply use the version defined in the superclass.

Any class that contains one or more abstract methods must also be declared abstract.

# There can be no objects of an abstract class.

# You cannot declare abstract constructors, or abstract static methods.

# You can declare static methods in abstract class.

Because there can be no objects for abstract class. If they had allowed to call abstract static methods,

it would that mean we are calling an empty method (abstract) through classname because it is static.

Any subclass of an abstract class must either implement all of the abstract methods in the superclass,or be declared abstract itself.

Abstract classes can include as much implementation as they see fit i.e.there can be concrete methods(methods with body) in abstract class.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references,

because Java’s approach to run-time polymorphism is implemented through the use of superclass references.

A public constructor on an abstract class doesn't make any sense because you can't instantiate an abstract class directly (can only instantiate through a derived type that itself is not marked as abstract)

Check: https://stackoverflow.com/questions/260666/can-an-abstract-class-have-a-constructor

**Abstract class vs Interface:**

Type of methods:

Interface can have only abstract methods.

Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.

Final Variables:

Variables declared in a Java interface are by default final.

An abstract class may contain non-final variables.

Type of variables:

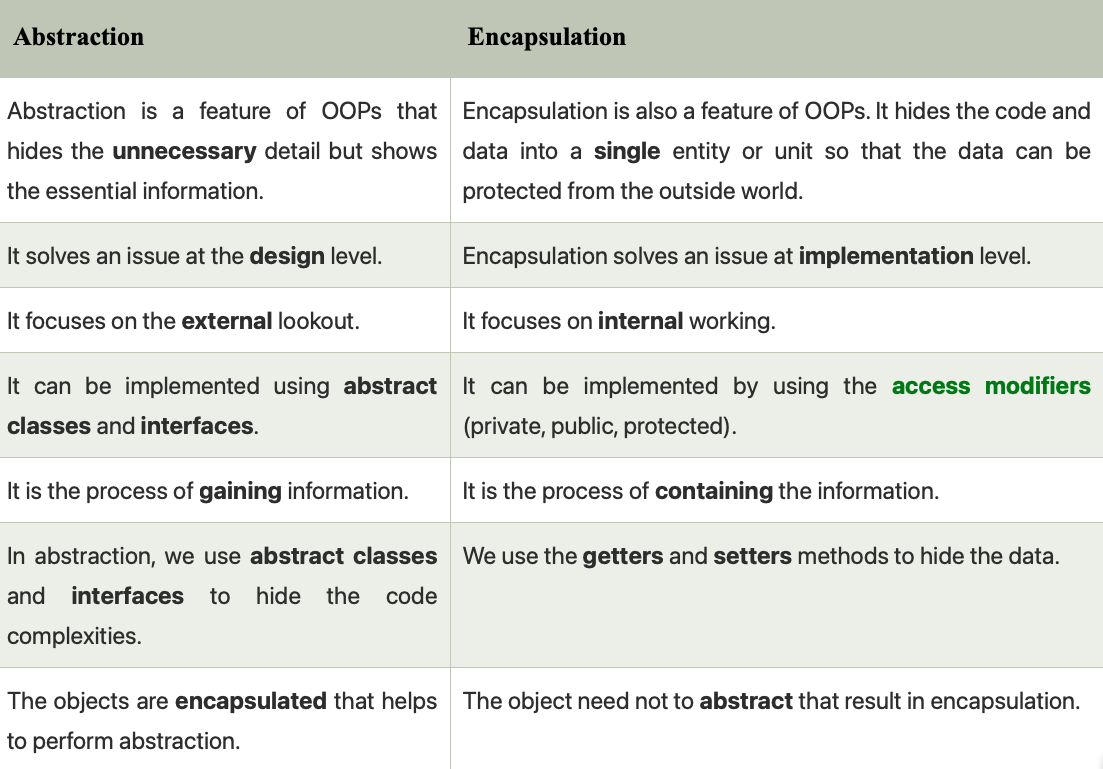
Abstract class can have final, non-final, static and non-static variables.

Interface has only static and final variables.

Implementation:

Abstract class can provide the implementation of interface.

Interface can’t provide the implementation of abstract class.

**Inheritance vs Abstraction**:

A Java interface can be implemented using keyword “implements”

and abstract class can be extended using keyword “extends”.

Multiple implementation:

An interface can extend another Java interface only,

an abstract class can extend another Java class and implement multiple Java interfaces.

Accessibility of Data Members:

Members of a Java interface are public by default.

A Java abstract class can have class members like private, protected, etc.

**The this Keyword:**

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword.

this can be used inside any method to refer to the current object. That is, this is always a reference to the object on

which the method was invoked.

**final** Keyword:

A field can be declared as final. Doing so prevents its contents from being modified, making it, essentially, a constant.

This means that you must initialize a final field when it is declared.

It is a common coding convention to choose all uppercase identifiers for final fields:

final int FILE\_OPEN = 2;

Unfortunately, final guarantees immutability only when instance variables are primitive types, not reference types.

If an instance variable of a reference type has the final modifier, the value of that instance variable (the reference

to an object) will never change—it will always refer to the same object—but the value of the object itself can change.

**The finalize( ) Method**: Sometimes an object will need to perform some action when it is destroyed.

To handle such situations, Java provides a mechanism called finalization. By using finalization,you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.To add a finalizer to a class, you simply define the finalize( ) method. The Java run time calls that method whenever it is about to recycle an object of that class. Right before an asset is freed, the Java run time calls the finalize( ) method on the object.

protected void finalize( ) {

// finalization code here

}

**Constructors**:

Once defined, the constructor is automatically called when the object is created, before the new operator completes.

Constructors look a little strange because they have no return type, not even void.

This is because the implicit return type of a class’ constructor is the class type itself.

In the line : Box mybox1 = new Box(); //new Box( ) is calling the Box( ) constructor.

**Inheritance and constructors** in Java:

In Java, constructor of base class with no argument gets automatically called in derived class constructor.

For example, output of following program given below is:

Base Class Constructor Called

Derived Class Constructor Called

// filename: Main.java

class Base {

Base() {

System.out.println("Base Class Constructor Called ");

} }

class Derived extends Base {

Derived() {

System.out.println("Derived Class Constructor Called ");

} }

public class Main {

public static void main(String[] args) {

Derived d = new Derived();

} }

Any class will have a default constructor, does not matter if we declare it in the class or not. If we inherit a class,

then the derived class must call its super class constructor. It is done by default in derived class.

If it does not have a default constructor in the derived class, the JVM will invoke its default constructor and call

the super class constructor by default. If we have a parameterised constructor in the derived class still it calls the

default super class constructor by default. In this case, if the super class does not have a default constructor,

instead it has a parameterised constructor, then the derived class constructor should call explicitly call the

parameterised super class constructor.

**INHERITENCE :**

To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword.

class subclass-name extends superclass-name { // body of class

}

You can only specify one superclass for any subclass that you create. Java does not support the inheritance of

multiple superclasses into a single subclass. You can, as stated, create a hierarchy of inheritance in which a subclass

becomes a superclass of another subclass. However, no class can be a superclass of itself.

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass

that have been declared as private.

A Superclass Variable Can Reference a Subclass Object:

It is important to understand that it is the type of the reference variable—not the type of the object that it refers

to—that determines what members can be accessed.

When a reference to a subclass object is assigned to a superclass reference variable, you will have access only to

those parts of the object defined by the superclass.

plainbox = weightbox;

(superclass) (subclass)

SUPERCLASS ref = new SUBCLASS(); // HERE ref can only access methods which are available in SUPERCLASS

Using **super**:

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.

super has two general forms. The first calls the superclass’ constructor.

The second is used to access a member of the superclass that has been hidden by a member of a subclass.

BoxWeight(double w, double h, double d, double m) {

super(w, h, d); // call superclass constructor

weight = m;

}

Here, BoxWeight( ) calls super( ) with the arguments w, h, and d. This causes the Box constructor to be called,

which initializes width, height, and depth using these values. BoxWeight no longer initializes these values itself.

It only needs to initialize the value unique to it: weight. This leaves Box free to make these values private if desired.

Thus, **super( ) always** refers to the superclass immediately above the calling class.

This is true even in a multileveled hierarchy.

class Box {

private double width;

private double height;

private double depth;

// construct clone of an object

Box(Box ob) { // pass object to constructor

width = ob.width;

height = ob.height;

depth = ob.depth;

}

}

class BoxWeight extends Box {

double weight; // weight of box

// construct clone of an object

BoxWeight(BoxWeight ob) { // pass object to constructor

super(ob);

weight = ob.weight;

}

}

Notice that super() is passed an object of type BoxWeight—not of type Box.This still invokes the constructor Box(Box ob).

NOTE: A superclass variable can be used to reference any object derived from that class.

Thus, we are able to pass a BoxWeight object to the Box constructor.Of course,Box only has knowledge of its own members.

A Second Use for super

The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in

which it is used.

super.member

Here, member can be either a method or an instance variable. This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

super( ) always refers to the constructor in the closest superclass. The super( ) in BoxPrice calls the constructor in

BoxWeight. The super( ) in BoxWeight calls the constructor in Box. In a class hierarchy, if a superclass constructor

requires parameters, then all subclasses must pass those parameters “up the line.” This is true whether or not a

subclass needs parameters of its own.

If you think about it, it makes sense that constructors complete their execution in order of derivation.

Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and

possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its execution first.

NOTE: If super( ) is not used in subclass' constructor, then the default or parameterless constructor of each superclass will be executed.

**Using final with Inheritance**:

The keyword final has three uses:

# First**, it can be used to create the equivalent of a named constant.**

**# Using final to Prevent Overriding:**

**To disallow a method from being overridden, specify final as a modifier at the start of its declaration**.

Methods declared as final cannot be overridden.

Methods declared as final can sometimes provide a performance enhancement: The compiler is free to inline calls to them because it “knows” they will not be overridden by a subclass. When a small final method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is an option only with final methods.

Normally, Java resolves calls to methods dynamically, at run time. This is called late binding. However, since final

methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

# **Using final to Prevent Inheritance:**

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final.

NOTE: Declaring a class as final implicitly declares all of its methods as final, too.

As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete

by itself & relies upon its subclasses to provide complete implementations.

# NOTE: **Although static methods can be** inherited ,there is no point in overriding them in child classes because the

method in parent class will run always no matter from which object you call it. That is why static interface methods

cannot be inherited because these method will run from the parent interface and no matter if we were allowed to

override them, they will always run the method in parent interface.

That is why static interface method must have a body.

NOTE **: Polymorphism does not apply to instance variables.**