Universal Class (Object Class)

- The Object class sits at the top of the class hierarchy tree in the Java development environment.
- Every class in the Java system is a descendent (direct or indirect) of the Object class.
- The Object class defines the basic state and behavior that all objects must have, such as the ability to compare one to another object, to convert to a string, to wait on a condition variable, to notify other objects that a condition variable has changed, and to return the object's class.
- parent class reference variable can refer the child class object, known as upcasting.

Example:

- getObject() method that returns an object but it can be of any type like Employee, Student etc, we can use Object class reference to refer that object.
  Object obj=getObject();

Methods of Object class

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long	causes the current thread to wait for the

timeout)throws InterruptedException	specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

1. **equals ():**

- Use the equals to compare two objects for equality. This method returns true if the objects are equal, false otherwise. Note that equality does not mean that the objects are the same object.
- **Example:**

```
Integer one = new Integer(1), anotherOne = new Integer(1);
if (one.equals(anotherOne))
        System.out.println("objects are equal");
```

- This code will display objects are equal even though one and anotherOne reference two different, distinct objects. They are considered equal because they contain the same integer value.

2. **getClass():**

- The getClass method is a final method (cannot be overridden) that returns a runtime representation of the class of this object. This method returns a Class object.
- **Example:**

```
void PrintClassName(Object obj) {
        System.out.println("The Object's class is " + obj.getClass().getName());
}
```

- One use of the getClass method is to create a new instance of a class without knowing what the class is at compile time. This sample method creates a new instance of the same class as obj which can be any class that inherits

from Object (which means that it could be any class)

- **Example:**

```
Object createNewInstanceOf(Object obj) {
        return obj.getClass().newInstance();
}
```

3. toString():

- Object's toString method returns a String representation of the object. You can use toString to display an object.
- For example, you could display a String representation of the current Thread like this:
- **Example:**

```
System.out.println(Thread.currentThread().toString());
```

- The String representation for an object is entirely dependent on the object. The String representation of an Integer object is the integer value displayed as text. The String representation of a Thread object contains various attributes about the thread, such as its name and priority.

- The Object class also provides five methods that are critical when writing multithreaded Java programs:
  - notify
  - notifyAll
  - wait (three versions)

Inheritance

- Reusability is another aspect of oop.Java supports reusability using inheritance.
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- In Java this is also called extending a class. One class can extend another class and thereby inherit from that class.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

- Inheritance represents the IS-A relationship which is also known as a parent-

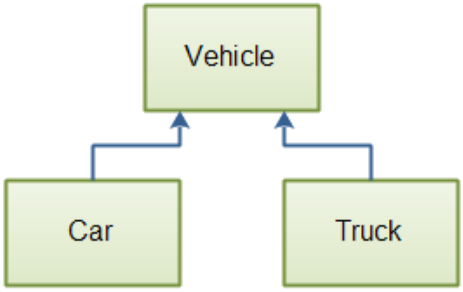
child relationship.

- When one class inherits from another class in Java, the two classes take on certain roles. The class that extends (inherits from another class) is the subclass and the class that is being extended (the class being inherited from) is the superclass . In other words, the subclass extends the superclass. Or, the subclass inherits from the superclass.
- Another commonly used term for inheritance is specialization and generalization. A subclass is a specialization of a superclass, and a superclass is a generalization of one or more subclasses.

Syntax:

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

- The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
- Inheritance can be an effective method to share code between classes that have some traits in common, yet allowing the classes to have some parts that are different.
- Here is diagram illustrating a class called Vehicle, which has two subclasses called Car and Truck.



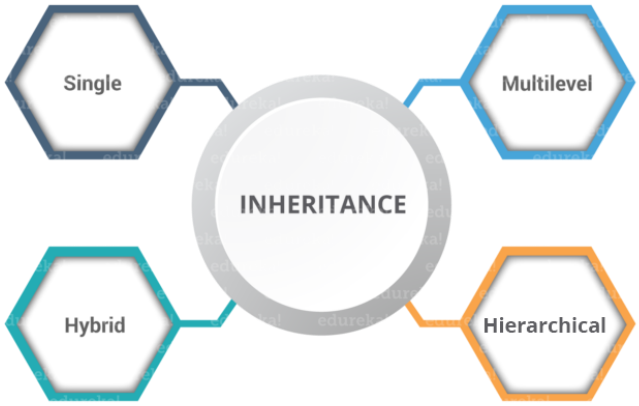
Example:

```
class Employee{
        float salary=40000;
}
class Programmer extends Employee{
        int bonus=10000;
         public static void main(String args[]){
                Programmer p=new Programmer();
```

```
        System.out.println("Programmer salary is:"+p.salary);
        System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Types of inheritance in java

- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only.



- Multiple inheritance is not supported in Java through class.
- When one class inherits multiple classes, it is known as multiple inheritance.

Single Inheritance

- In single inheritance, one class inherits the properties of another. It enables a derivedclass to inherit the properties and behavior from a single parent class.
- This will, in turn, enable code reusability as well as add new features to the existing code.

Here, Class A is your parent class and Class B is your child class which inherits the properties and behavior of the parent class. A similar concept is represented in the below code:

Example:

```
class Animal{
        void eat(){System.out.println("eating");}
}
class Dog extends Animal{
        void bark(){System.out.println("barking");}
}
class TestInheritance{
public static void main(String args[]){
          Dog d=new Dog();
          d.bark();
          d.eat();
}
}
```

Multi-level Inheritance

- When a class is derived from a class which is also derived from another class, i.e. a class having more than one parent class but at different levels, such type of inheritance is called Multilevel Inheritance.

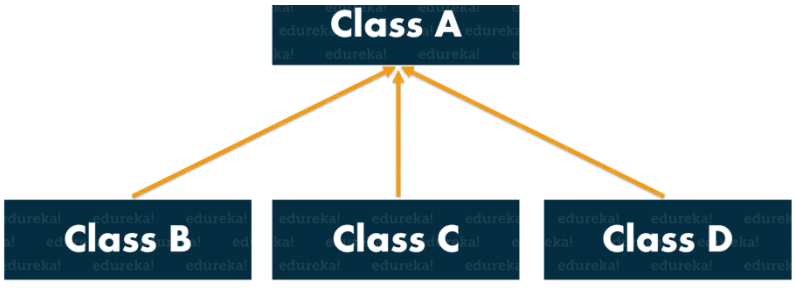
- If we talk about the flowchart, class B inherits the properties and behavior of class A and class C inherits the properties of class B.
- Here A is the parent class for B and class B is the parent class for C. So in this case class C implicitly inherits the properties and methods of class A along with Class B. That's what is multilevel inheritance.

Example:

```
class Animal{
        void eat(){System.out.println("eating…");}
}
class Dog extends Animal{
        void bark(){System.out.println("barking…");}
}
class Puppy extends Dog{
        void weep(){System.out.println("weeping…");}
}
class TestInheritance2{
       public static void main(String args[]){
                Puppy d=new Puppy();
                d.weep();
                d.bark();
                d.eat();
}
}
```

Hierarchical Inheritance

- When a class has more than one child classes (subclasses) or in other words, more than one child classes have the same parent class, then such kind of inheritance is known as hierarchical.



- In the above flowchart, Class B and C are the child classes which are inheriting from the parent class i.e Class A.

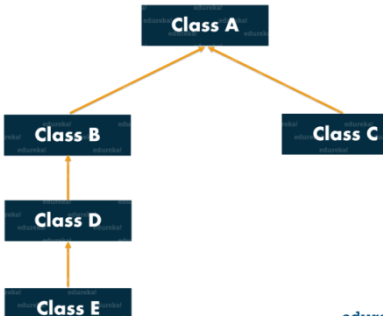
Example:

```
class Animal{
        void eat(){System.out.println("eating…");}
}
```

```
class Dog extends Animal{
        void bark(){System.out.println("barking…");}
}
class Cat extends Animal{
       void meow(){System.out.println("meowing…");}
}
class TestInheritance3{
        public static void main(String args[]){
                Cat c=new Cat();
                c.meow();
                c.eat();
        }
}
```

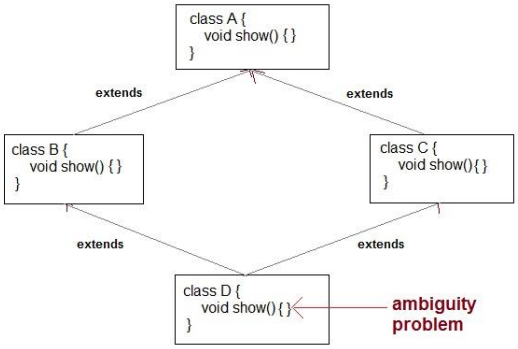
Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance.



Rules of Inheritance in Java

RULE 1: Multiple Inheritance is NOT permitted in Java.



- Multiple inheritance refers to the process where one child class tries to extend more than one parent class.
- In the above example Class A is a parent class for Class B and C, which are further extended by class D. This is results in Diamond Problem. Why? Say you have a method show() in both the classes B and C, but with different functionalities.
- When Class D extends class B and C, it automatically inherits the characteristics of B and C including the method show(). Now, when you try to invoke show() of class B, the compiler will get confused as to which show() to be invoked (either from class B or class C). Hence it leads to ambiguity.

Example:

```
class A{
        void msg(){System.out.println("Hello");}
}
class B{
        void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were
        public static void main(String args[]){
                C obj=new C();
                obj.msg();//Now which msg() method would be invoked?
        }
}
```

RULE 2: Cyclic Inheritance is NOT permitted in Java.

- It is a type of inheritance in which a class extends itself and form a loop itself. Now think if a class extends itself or in any way, if it forms cycle within the user-defined classes, then is there any chance of extending the Object class. That is the reason it is not permitted in Java.

Example:

```
class Demo1 extends Demo2
{
    //code here
}
class Demo2 extends Demo1
{
    //code here
}
```

RULE 3: Private members do NOT get inherited

RULE 4: Constructors cannot be Inherited in Java.

- A constructor cannot be inherited, as the subclasses always have a different name.

```
class A {
  A();
}
 class B extends A{
  B();
}
```

You can do only:

```
B b = new B();  // and not new A()
```

- Methods, instead, are inherited with "the same name" and can be used. You can still use constructors from A inside B's implementation though:

```
class B extends A{
  B() {
     super();
   }
}
```

RULE 5: In Java, we assign parent reference to child objects.

- Parent is a reference to an object that happens to be a subtype of Parent, i.e.a Child Object.

Constructors and Inheritance

- Constructor of sub class is invoked when we create the object of subclass; it by default invokes the default constructor of super class.
- Hence, in inheritance the objects are constructed top-down. The super class constructor can be called explicitly using the super keyword, but it should be first statement in a constructor.
- The super keyword refers to the super class, immediately above of the calling class in the hierarchy. The use of multiple super keywords to access an ancestor class other than the direct parent is not permitted.

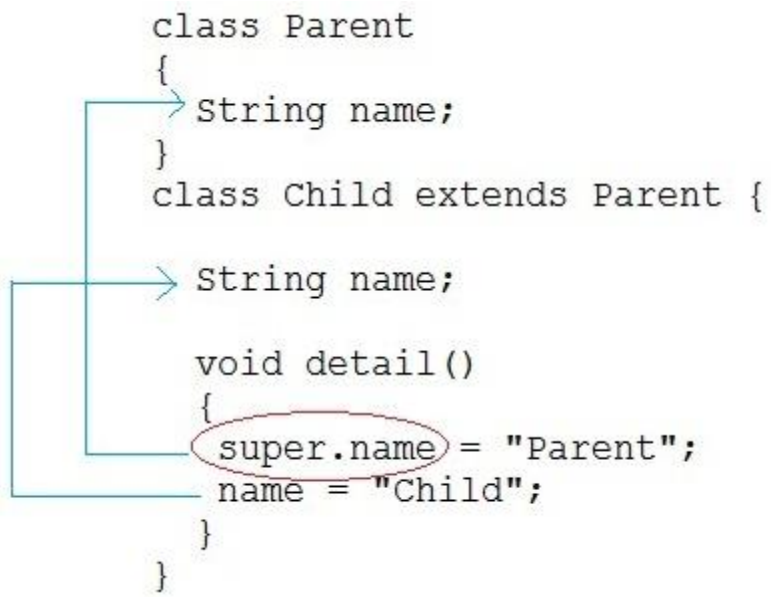
Example:

```
class ParentClass{
    ParentClass(){
        System.out.println("Constructor of Parent");
  }
```

```
}
class JavaExample extends ParentClass{
  JavaExample(){
        System.out.println("Constructor of Child");
  }
  public static void main(String args[]){
        //Creating the object of child class
        new JavaExample();
  }
}
```

super keyword

- In Java, super keyword is used to refer to immediate parent class of a child class. In other words **super** keyword is used by a subclass whenever it need to refer to its immediate super class.



Usage of Java super Keyword

1. **super can be used to refer immediate parent class instance variable.**

- We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

  Syntax: super.member;

  Example:

  class Animal{

```
String color="white";
}
class Dog extends Animal{
        String color="black";
        void printColor(){
                System.out.println(color);//prints color of Dog class
                System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
        public static void main(String args[]){
                Dog d=new Dog();
                d.printColor();
}}
```

2. super can be used to invoke parent class method

- The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

Syntax: super.methodname(paramlist);

Example:

```
class Animal{
    void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
    void eat(){System.out.println("eating bread...");}
    void bark(){System.out.println("barking...");}
     void work(){
        super.eat();
        bark();
    }
}
class TestSuper2{
     public static void main(String args[]){
            Dog d=new Dog();
            d.work();
}    }
```

3. **super() can be used to invoke immediate parent class constructor.**

- The super keyword can also be used to invoke the parent class constructor

Syntax: super(paramlist);

Example:

```
class Animal{
    Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
    Dog(){
        super();
        System.out.println("dog is created");
    }
}
class TestSuper3{
    public static void main(String args[]){
        Dog d=new Dog();
}}
```

- super() is added in each class constructor automatically by compiler if there is no super() or this().
- As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

Example:

```
class Person{
    int id;
    String name;
    Person(int id,String name){
        this.id=id;
        this.name=name;
    }
}
class Emp extends Person{
    float salary;
    Emp(int id,String name,float salary){
        super(id,name);//reusing parent constructor
        this.salary=salary;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
```

```
}
class TestSuper5{
        public static void main(String[] args){
                Emp e1=new Emp(1,"ankit",45000f);
                e1.display();
}}
```

Can you use both this() and super() in a Constructor?

- NO, because both super() and this() must be first statement inside a constructor. Hence we cannot use them together.

Method Overriding

- When we declare the same method in child class which is already present in the parent class the this is called method overriding.
- In this case when we call the method from child class object, the child class version of the method is called. However we can call the parent class method using super keyword.

Example:

```
class ParentClass{
  ParentClass(){
        System.out.println("Constructor of Parent");
  }
  void disp(){
        System.out.println("Parent Method");
  }
}
class JavaExample extends ParentClass{
  JavaExample(){
        System.out.println("Constructor of Child");
  }
  void disp(){
        System.out.println("Child Method");
        super.disp();
  }
  public static void main(String args[]){
        JavaExample obj = new JavaExample();
        obj.disp();
```

```
  }
}
```

Interface in java:

- An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.
- The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
- An interface is similar to a class in the following ways –
  - An interface can contain any number of methods.
  - An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
  - The byte code of an interface appears in a .class file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.
- an interface is different from a class in several ways, including –
  - You cannot instantiate an interface.
  - An interface does not contain any constructors.
  - All of the methods in an interface are abstract.
  - An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
  - An interface is not extended by a class; it is implemented by a class.
  - An interface can extend multiple interfaces.
  - It cannot be instantiated just like the abstract class.

How to declare 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 abstract
```

```
   // by default.
}
```

Internal addition by the compiler

- The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.
- In other words, Interface fields are public, static and final by default, and the methods are public and abstract.
- Interfaces have the following properties –
  - An interface is implicitly abstract. You do not need to use the abstract keyword while declaring an interface.
  - Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
  - Methods in an interface are implicitly public.

```
/* File name : Animal.java */
interface Animal {
  public void eat();
  public void travel();
}
```

Implementing Interfaces

- When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface.
- If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.
- A class uses the implements keyword to implement an interface

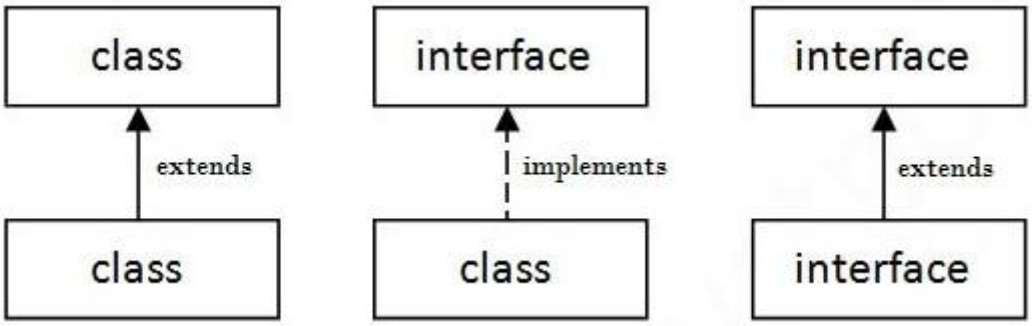
Example:

```
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();
         }
}
```

- When overriding methods defined in interfaces, there are several rules to be followed –
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so, interface methods need not be implemented.
- When implementation interfaces, there are several rules –
  - A class can implement more than one interface at a time.
  - A class can extend only one class, but implement many interfaces.
  - An interface can extend another interface, in a similar way as a class can extend another class.

The relationship between classes and interfaces

- As shown in the figure given below, a class extends another class, an interface extends another interface, but a class implements an interface.



Extending Interfaces

- An interface can extend another interface in the same way that a class can extend another class. The extends keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

Example

```
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();
        }
}
```

Extending Multiple Interfaces

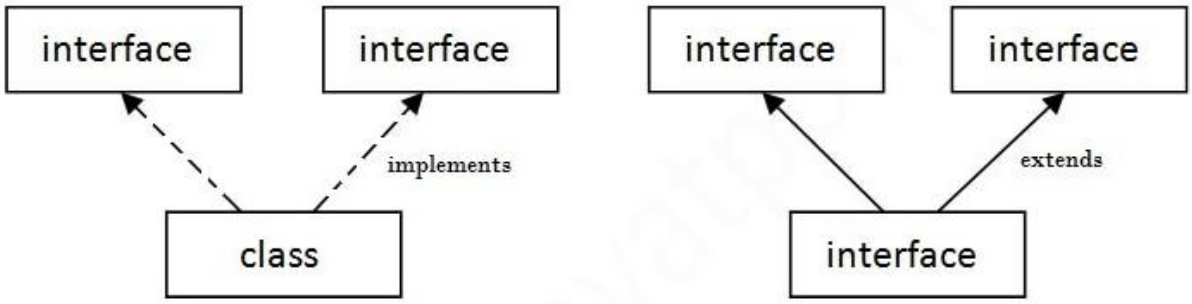
- A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.
- The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

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();
        }
}
```

Multiple inheritance in Java by interface

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



Multiple Inheritance in Java

What is marker or tagged interface?

- The most common use of extending interfaces occurs when the parent interface does not contain any methods.
- An interface which has no member or method is known as a 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.
- For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as –

Example

```
package java.util;
public interface EventListener
{}
```

- There are two basic design purposes of tagging interfaces –
- ➢ Creates a common parent –
- As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces.
- For example, when an interface extends Event Listener, the JVM knows that this particular interface is going to be used in an event delegation scenario.
- ➢ Adds a data type to a class –
- This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

Nested Interface in Java

- An interface can have another interface which is known as a nested interface

Example:

```
interface printable{
        void print();
         interface MessagePrintable{
         void msg();
         }
}
```

Why Use Nested Classes?

- Compelling reasons for using nested classes include the following:
- **It is a way of logically grouping classes that are only used in one place:**
  - If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- **It increases encapsulation:**
  - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- **It can lead to more readable and maintainable code:**
  - Nesting small classes within top-level classes places the code closer to where it is used.

Nested Classes

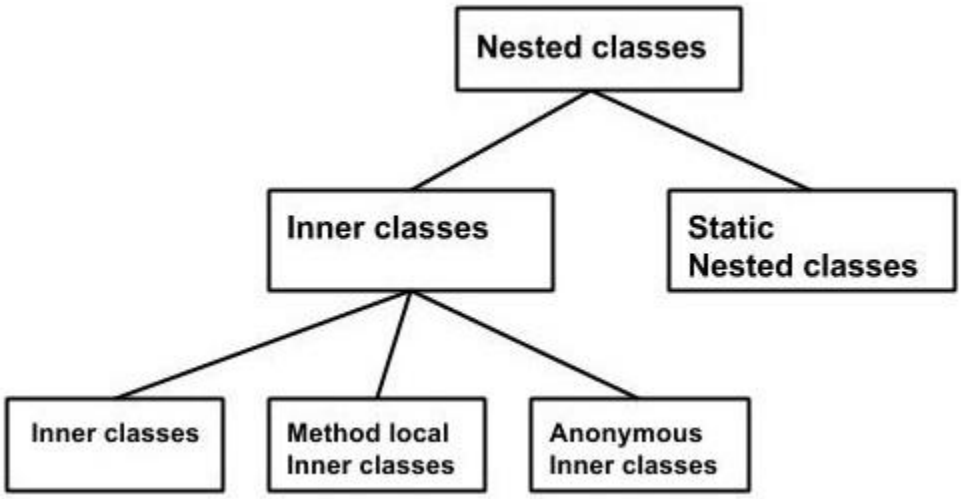
- In Java, just like methods, variables of a class too can have another class as its member. Writing a class within another is allowed in Java. The class written within is called the nested class, and the class that holds the inner class is called the outer class.
- Here, the class Outer_Demo is the outer class and the class Inner_Demo is the nested class.

Syntax:

```
class Outer_Demo {
  class Inner_Demo {
  }
}
```

Nested classes are divided into two types –

- Non-static nested classes – These are the non-static members of a class.
- Static nested classes – These are the static members of a class.



Java static nested class

- A static inner class is a nested class which is a static member of the outer class.
- It can be accessed without instantiating the outer class, using other static members.
- It can be accessed by outer class name. It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

Syntax:

```
class MyOuter {
  static class Nested_Demo {
  }
}
```

Example:

```
class TestOuter1{
 static int data=30;
 static class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
 TestOuter1.Inner obj=new TestOuter1.Inner();
 obj.msg();
 }
```

```
}
```

Inner Classes (Non-static Nested Classes)

- Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier private, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.
- Inner classes are of three types depending on how and where you define them. They are –
  1. Member Inner Class
  2. Method-local Inner Class
  3. Anonymous Inner Class

1. Member Inner Class

- A non-static Nested class that is created outside a method is called Member inner class. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Example:

```
class Outer_Demo {
  int num;
  private class Inner_Demo {
    public void print() {
      System.out.println("This is an inner class");
    }
  }
  void display_Inner() {
     Inner_Demo inner = new Inner_Demo();
     inner.print();
  }
 }
public class My_class {
  public static void main(String args[]) {
    Outer_Demo outer = new Outer_Demo();
    outer.display_Inner();
  }
}
```

Accessing the Private Members

- inner classes are also used to access the private members of a class.
- Suppose, a class is having private members to access them. Write an inner class in it, return the private members from a method within the inner class, say, getValue(), and finally from another class (from which you want to access the private members) call the getValue() method of the inner class.
- Instantiate the inner class, initially you have to instantiate the outer class. Thereafter, using the object of the outer class, following is the way in which you can instantiate the inner class.

Syntax:

```
Outer_Demo outer = new Outer_Demo();
Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
```

Example:

```
class Outer_Demo {
      private int num = 175;
      public class Inner_Demo {
         public int getNum() {
                  System.out.println("This is the getnum method of the inner class");
                  return num;
         }
    }
 }
 public class My_class2 {
   public static void main(String args[]) {
         Outer_Demo outer = new Outer_Demo();
         Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
        System.out.println(inner.getNum());
   }
 }
```

2. Method local inner class:

- A non-static Nested class that is created inside a method is called local inner class.
- If you want to invoke the methods of local inner class, you must instantiate this class inside the method.
- We cannot use private, public or protected access modifiers with local inner class. Only abstract and final modifiers are allowed.

Example:

```
class Outer
{
        int count;
        public void display()
        {
                for(int i=0;i<5;i++)
                {
                        class inner
                        {
                                public void show()
                                {
                                        System.out.println("Inside inner "+(count++));
                                }
                        }
                        Inner in=new Inner();
                        in.show();
                }
        }
}
class Test
{
        public static void main(String[] args)
        {
                Outer ot = new Outer();
                ot.display();
        }
}
```

Example:

- Inner class instantiated outside Outer class

```
class Outer
{
        int count;
        public void display()
        {
                Inner in = new Inner();
```

```
                in.show();
        }
        class Inner
        {
                public void show()
                {
                        System.out.println("Inside inner "+(++count));
                }
        }
}
class Test
{
        public static void main(String[] args)
        {
                Outer ot = new Outer();
                Outer.Inner in = ot.new Inner();
                in.show();
        }
}
```

3. Annonymous class:

- A class without any name is called Annonymous class.
- Anonymous classes in Java are nested classes without a class name. They are typically declared as either subclasses of an existing class, or as implementations of some interface.
- A class created for implementing interface or extending class. Its name is decided by the java compiler.
- In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface.

Syntax:

```
AnonymousInner an_inner = new AnonymousInner() {
  public void my_method() {
    ........
    ........
  }
```

```
};
```

- Java Anonymous inner class can be created by two ways:
  - Class (may be abstract or concrete).
  - Interface

➢ **Java anonymous inner class example using class**

```
abstract class Person{
 abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
  void eat(){System.out.println("nice fruits");}
  };
  p.eat();
 }
}
```

- A class is created but its name is decided by the compiler which extends the Person class and provides the implementation of the eat() method.
- An object of Anonymous class is created that is referred by p reference variable of Person type.

Example:

Internal class generated by the compiler

```
import java.io.PrintStream;
static class TestAnonymousInner$1 extends Person
{
  TestAnonymousInner$1(){}
  void eat()
   {
      System.out.println("nice fruits");
   }
}
```

Java anonymous inner class example using interface

```
interface Eatable{
 void eat();
}
class TestAnnonymousInner1{
 public static void main(String args[]){
```

```
Eatable e=new Eatable(){
 public void eat(){System.out.println("nice fruits");}
};
e.eat();
}
}
```

- A class is created but its name is decided by the compiler which implements the Eatable interface and provides the implementation of the eat() method.
- An object of Anonymous class is created that is referred by p reference variable of Eatable type.

Example:

Internal class generated by the compiler

```
import java.io.PrintStream;
static class TestAnonymousInner1$1 implements Eatable
{
    TestAnonymousInner1$1(){}
    void eat(){System.out.println("nice fruits");}
}
```

Anonymous Inner Class as Argument

- Generally, if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method.
- But in all the three cases, you can pass an anonymous inner class to the method. Here is the syntax of passing an anonymous inner class as a method argument –

```
obj.my_Method(new My_Class() {
  public void Do() {
    .....
    .....
  }
});
// interface
interface Message {
  String greet();
}
```

```
public class My_class {
  public void displayMessage(Message m) {
    System.out.println(m.greet()+", This is an example of anonymous inner class as
an argument");
  }
  public static void main(String args[]) {
        My_class obj = new My_class();
         obj.displayMessage(new Message() {
                  public String greet() {
                           return "Hello";
                  }
         });
  }
}
```

Concrete class

- A class which is not abstract is referred as Concrete class.

Abstract class in Java

- A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).
- It shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
- Abstraction lets you focus on what the object does instead of how it does it.
- 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.

Ways to achieve Abstraction

- There are two ways to achieve abstraction in java

- Abstract class (0 to 100%)
- Interface (100%)

When to use Abstract Methods & Abstract Class?

- Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same Abstract class and provide different implementations for the abstract methods.
- Abstract classes are used to define generic types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

Syntax:

```
abstract class classname{
        ……..
}
```

Why can't we create the object of an abstract class?

- Because these classes are incomplete, they have abstract methods that have no body so if java allows you to create object of this class then if someone calls the abstract method using that object then What would happen?There would be no actual implementation of the method to invoke.An abstract class is like a template, so you have to extend it and build on it before you can use it.

Abstract Method in Java

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

Syntax:

```
abstract returntype methodname();//no method body and abstract
```

Example:

```
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();
```

```
        }
}
```

Difference between abstract class and interface

- Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.
- But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

abstract class achieves partial abstraction (0 to 100%)	interface achieves fully abstraction (100%).

Example:
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Final class:

- The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:
  1. variable
  2. method
  3. class
- The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

1. final variable

- If you make any variable as final, you cannot change the value of final variable once it is initialized(It will be constant).

Example:

```
class Bike9{
        final int speedlimit=90;//final variable
        void run(){
                speedlimit=400;//can't change value of final variable gives error
        }
        public static void main(String args[]){
                Bike9 obj=new  Bike9();
                obj.run();
        }
}
```

Blank final variable

- A final variable that is not initialized at the time of declaration is known as blank final variable**.** We must initialize the blank final variable in constructor of the class otherwise it will throw a compilation error

Example:

```
class Demo{
        final int MAX_VALUE;
        Demo(){
                MAX_VALUE=100;//It must be initialized in constructor
        }
        void myMethod(){
                System.out.println(MAX_VALUE);
        }
        public static void main(String args[]){
                Demo obj=new  Demo();
                obj.myMethod();
        }
}
```

Uninitialized static final variable

- A static final variable that is not initialized during declaration can only be initialized in static block.

Example:

```
class Example{
    static final int ROLL_NO;
    static{
        ROLL_NO=1230;
    }
    public static void main(String args[]){
        System.out.println(Example.ROLL_NO);
  }
}
```

2. final method

- If you make any method as final, you cannot override it.
- Which means even though a sub class can call the final method of parent class without any issues but it cannot override it.
- As mentioned previously, the final modifier prevents a method from being modified in a subclass.
- The main intention of making a method final would be that the content of the method should not be changed by any outsider.

Example:

```
class Bike{
 final void run(){System.out.println("running");}
}
class Honda extends Bike{
  void run(){System.out.println("running safely with 100kmph");}   //gives error
  public static void main(String args[]){
  Honda honda= new Honda();
  honda.run();
  }
}
```

3. Final Class

- The main purpose of using a class being declared as final is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

Example:

```
final class XYZ{
}
class ABC extends XYZ{
  void demo(){
    System.out.println("My Method");
  }
  public static void main(String args[]){
    ABC obj= new ABC();
    obj.demo();
  }
}
```

What is final parameter?

- If you declare any parameter as final, you cannot change the value of it.

Example:

```
class Bike11{
 int cube(final int n){
  n=n+2;//can't be changed as n is final
  n*n*n;
 }
 public static void main(String args[]){
```

```
    Bike11 b=new Bike11();
    b.cube(5);
  }
}
```

Can we declare a constructor final?
No, because constructor is never inherited.

Some points for final keyword:
1)A constructor cannot be declared as final.
2) Local final variable must be initializing during declaration.
3) All variables declared in an interface are by default final.
4) We cannot change the value of a final variable.
5) A final method cannot be overridden.
6) A final class not be inherited.
7) If method parameters are declared final then the value of these parameters cannot be changed.
8) It is a good practice to name final variable in all CAPS.
9) final, finally and finalize are three different terms. finally is used in exception handling and finalize is a method that is called by JVM during garbage collection.

Difference between abstract and final class

PROPERTY	ABSTRACT CLASS	FINAL CLASS
SUBCLASSING	Should be subclassed to override the functionality of abstract methods	Can never be subclassed as final does not permit
METHOD ALTERATIONS	Abstract class methods functionality can be altered in subclass	Final class methods should be used as it is by other classes
INSTANTIATION	Cannot be instantiated	Can be instantiated
OVERRIDING CONCEPT	For later use, all the abstract methods should be overridden	Overriding concept does not arise as final class cannot be inherited

INHERITANCE	Can be inherited	Cannot be inherited
ABSTRACT METHODS	Can contain abstract methods	Cannot contain abstract methods
PARTIAL IMPLEMENTATION	A few methods can be implemented and a few cannot	All methods should have implementation
IMMUTABLE OBJECTS	Cannot create immutable objects (infact, no objects can be created)	Immutable objects can be created (eg. String class)
NATURE	It is an incomplete class (for this reason only, designers do not allow to create objects)	It is a complete class (in the sense, all methods have complete functionality or meaning)
ADDING EXTRA FUNCTIONALITY	Extra functionality to the methods can be added in subclass	No extra functionality can be added and should be used as it is

Normal import and Static Import

- We can use an import statement to import classes and interface of a particular package**.**
- Whenever we are using import statement it is not required to use the fully qualified name and we can use short name directly**.**
- We can use static import to import static member from a particular class and package**.** Whenever we are using static import it is not required to use the class name to access static member and we can use directly**.**

import statement

- To access a class or method from another package we need to use the **fully qualified name** or we can use **import** statements.
- The class or method should also be accessible. Accessibility is based on the **access modifiers**.
- **Private** members are accessible only within the same class. So we won't be able to access a private member even with the fully qualified name or an import statement.
- The **java.lang** package is automatically imported into our code by Java.

Example

```
import java.util.Vector;
```

```
public class ImportDemo {
  public ImportDemo() {
          Vector v = new Vector();
           v.add("DSCCS");
           v.add("BCA");
           v.add("Sem-4");
           System.out.println("Vector values are: "+ v);
            java.util.ArrayList list = new java.util.ArrayList();
            list.add("Colleges");
            list.add("Courses");
            System.out.println("Array List values are: "+ list);
   }
  public static void main(String arg[]) {
        new ImportDemo();
  }
}
```

Static Import Statement

- Static imports will import all static data so that can use without a class name.
- A static import declaration has two forms, one that imports a particular static member which is known as single static import and one that imports all static members of a class which is known as a static import on demand.

Syntax:

Single-static-import declaration:

import static <<package name>>.<<type name>>.<<static member name>>;

Static-import-on-demand declaration:

import static <<package name>>.<<type name>>.*;

- Static imports introduced in Java5 version.
- One of the advantages of using static imports is reducing keystrokes and re-usability.

Example:

```
import static java.lang.System.*; //Using Static Import
public class StaticImportDemo {
  public static void main(String args[]) {
       out.println("Welcome to JAVA");  }}
```

Advantage of static import:

- Less coding is required if you have access any static member of a class oftenly.

Disadvantage of static import:

- If you overuse the static import feature, it makes the program unreadable and unmaintainable.

What is the difference between import and static import?

- The import allows the java programmer to access classes of a package without package qualification whereas the static import feature allows accessing the static members of a class without the class qualification.
- The import provides accessibility to classes and interface whereas static import provides accessibility to static members of the class.

Static import rules

The following are some important rules about static import declaration.

1) If two static members with the same simple name are imported, one using single-static import declaration and other using static-import-on-demand declaration, the one imported using single-static import declaration takes precedence.

   Example:

```
import static package1.Class1.methodA; // Imports Class1.methodA() method
import static package2.Class2.*;  // Imports Class2.methodA() method too
public class Test {
    public static void main(String[] args) {
        methodA();   // Class1.methodA() will be called
    }
}
```

2) Using single-static-import declaration to import two static members with the same simple name is not allowed. The following static import declarations generate an error because both of them import the static member with the same simple name of methodA:

   Example:

```
import static package1.Class1.methodA;
import static package1.Class2.methodA; // An error
```

3) If a static member is imported using a single-static import declaration and there exists a static member in the same class with the same name, the static member in the class is used.

   Example1:

```
package package1;
```

```
 public class A {
    public static void test() {
        System.out.println("package1.A.test()");
    }
}
  // Test.java
package package2;
import static package1.A.test;
public class Test {
    public static void main(String[] args) {
        test(); // Will use package2.Test.test() method, not package1.A.test() method
    }
    public static void test() {
        System.out.println("package2.Test.test()");
    }
}
```

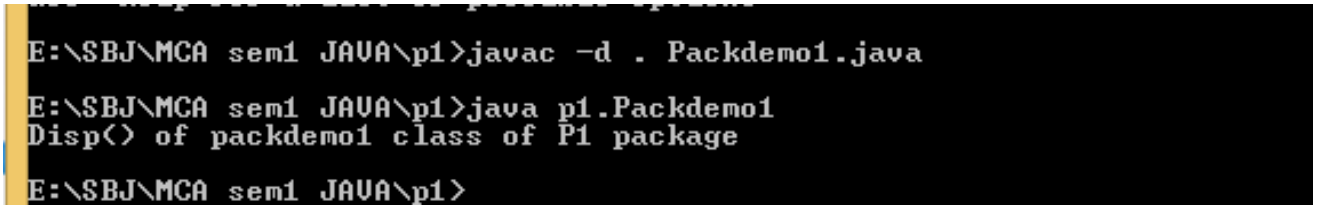
Output:

```
package2.Test.test()
```

Example2:

```
package p1;
class Packdemo1
{
    void disp()
    {
        System.out.println("Disp() of packdemo1 class of P1 package");
    }
    public static void main(String args[])
    {
        Packdemo1 p=new Packdemo1();
        p.disp();
    }
}
```

Output:



Example3:

Pack1_cl.java

```
package pack1;
public class Pack1_cl
{
        void disp()
        {
                System.out.println("Method of Pack1_cl class in pack1 with default access specifier");
        }
        private void disp1()
        {
                System.out.println("Method of Pack1_cl class in pack1 with private access specifier");
        }
        protected void disp2()
        {
                System.out.println("Method of Pack1_cl class in pack1 with protected access specifier");
        }
        public void disp3()
        {
                System.out.println("Method of Pack1_cl class in pack1 with public access specifier");
        }
        public static void main(String args[])
        {
                Pack1_cl p =new Pack1_cl();
                p.disp();
                p.disp1();
```

```
                p.disp2();
                p.disp3();
        }
}
```

Output:

```
E:\SBJ\MCA sem1 JAVA>javac pack1/Pack1_cl.java

E:\SBJ\MCA sem1 JAVA>java pack1.Pack1_cl
Method of Pack1_cl class in pack1 with default access specifier
Method of Pack1_cl class in pack1 with private access specifier
Method of Pack1_cl class in pack1 with protected access specifier
Method of Pack1_cl class in pack1 with public access specifier

E:\SBJ\MCA sem1 JAVA>_
```

Pack2_cl.java:

```
package pack2;
import pack1.*;
public class Pack2_cl
{
        public static void main(String args[])
        {
                Pack1_cl p =new Pack1_cl();
                //p.disp();
                //p.disp1();
                //p.disp2();
                p.disp3();
        }
}
```

Output:

```
E:\SBJ\MCA sem1 JAVA>java pack2.Pack2_cl
Method of Pack1_cl class in pack1 with public access specifier
```