UNIT – II

Inheritance, Packages and Interfaces – Hierarchical abstractions, Base class object, subclass, subtype, substitutability, forms of inheritance specialization, specification, construction, extension, limitation, combination, benefits of inheritance, costs of inheritance. Member access rules, super uses, using final with inheritance, polymorphism- method overriding, abstract classes, the Object class. Defining, Creating and Accessing a Package, Understanding CLASSPATH, importing packages, differences between classes and **interfaces**, defining an interface, implementing interface, applying interfaces, variables in interface and extending interfaces. Exploring java.io.

INHERITANCE IN JAVA

- Inheritance is an important pillar of OOP(Object-Oriented Programming).
- The process of obtaining the data members and methods from one class to another class is known as **inheritance**.

Important Terminologies Used in Java Inheritance

Super Class/Parent Class: The class whose features are inherited is known as a superclass(or a base class or a parent class).

Sub Class/Child Class: The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

Why Do We Need Java Inheritance?

Code Reusability: The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.

Method Overriding: Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.

Abstraction: The concept of abstract where we do not have to provide all details is achieved through inheritance. Abstraction only shows the functionality to the user.

How to Use Inheritance in Java?

- The **extends** keyword is used for inheritance in Java.
- Using the extends keyword indicates you are derived from an existing class. In other words, "extends" refers to increased functionality.

Syntax

```
class SubclassName extends SuperclassName
{
    //methods and fields
}
```

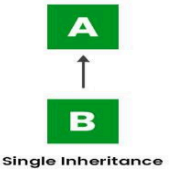
TYPES OF INHERITANCE

Based on number of ways inheriting the feature of base class into derived class we have five types of inheritance they are:

1. Single inheritance
2. Multilevel inheritance
3. Hierarchical inheritance
4. Multiple inheritance
5. Hybrid inheritance

1.Single inheritance

In single inheritance there exists single base class and single derived class.



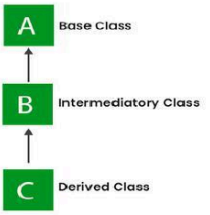
Single Inheritance

Example

```
class Animal
{
    String name;
    void show()
    {
            System.out.println("Animal name is:"+name);
    }
}
class Dog extends Animal
{
    void bark()
    {
            System.out.println("Barking");
    }
}
class TestInheritance
{
    public static void main(String args[])
    {
            Dog d=new Dog();
            d.name="DOG";
            d.show();
            d.bark();
    }
}
```

2. Multilevel inheritances in Java

- When there is a chain of inheritance, it is known as multilevel inheritance.
- In Multilevel inheritances there exists single base class, single derived class and multiple intermediate base classes.



Multilevel Inheritance

Example

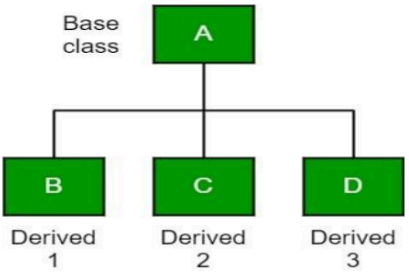
In the example, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
class Animal
{
	String name;
	void show()
	{
		System.out.println("Animal Name is"+name);
	}
}
class Dog extends Animal
{
	void bark()
	{
		System.out.println("Mother Dog Barking...");
	}
}
class BabyDog extends Dog
{
	void weep()
	{
		System.out.println("Baby Dog weeping");
	}
}
class TestInheritance2
{
	public static void main(String args[])
	{
		BabyDog d=new BabyDog();
		d.name="MotherDog";
		d.show();
		d.bark();
		d.weep();

	}
}
```

3. Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived class B, C and D.

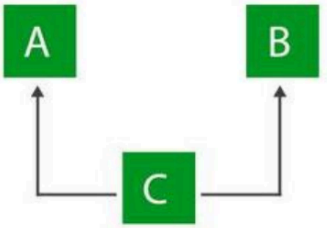


Example

```
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();
		//c.bark();//C.T.Error
	}
}
```

4. Multiple inheritance

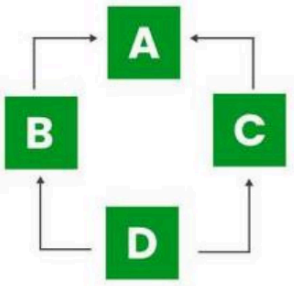
In multiple inheritance there exist multiple classes and single derived class.



The concept of multiple inheritance is not supported in java through concept of classes but it can be supported through the concept of interface.

5. Hybrid inheritance

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces.



Hybrid Inheritance

SUBSTITUTABILITY

- The inheritance concept used for the number of purposes in the java programming language. One of the main purposes is substitutability.
- The substitutability means that when a child class acquires properties from its parent class, the object of the parent class may be substituted with the child class object.
- For example, if B is a child class of A, anywhere we expect an instance of A we can use an instance of B.
- The substitutability can achieve using inheritance, whether using extends or implements keywords.

FORMS OF INHERITANCE

The following are the different forms of inheritance in java.

- Specialization
- Specification
- Construction
- Extension
- Limitation
- Combination

Specialization

It is the most ideal form of inheritance. The subclass is a special case of the parent class. It holds the principle of substitutability.

Specification

This is another commonly used form of inheritance. In this form of inheritance, the parent class just specifies which methods should be available to the child class but doesn't implement them. The java provides concepts like abstract and interfaces to support this form of inheritance. It holds the principle of substitutability.

Construction

This is another form of inheritance where the child class may change the behavior defined by the parent class (overriding). It does not hold the principle of substitutability.

Extension

This is another form of inheritance where the child class may add its new properties. It holds the principle of substitutability.

Limitation

This is another form of inheritance where the subclass restricts the inherited behavior. It does not hold the principle of substitutability.

Combination

This is another form of inheritance where the subclass inherits properties from multiple parent classes. Java does not support multiple inheritance type.

BENEFITS OF INHERITANCE

- Inheritance helps in code reuse. The child class may use the code defined in the parent class without re-writing it.
- Inheritance can save time and effort as the main code need not be written again.
- Inheritance provides a clear model structure which is easy to understand.
- An inheritance leads to less development and maintenance costs.
- With inheritance, we will be able to override the methods of the base class so that the meaningful implementation of the base class method can be designed in the derived class. An inheritance leads to less development and maintenance costs.
- In inheritance base class can decide to keep some data private so that it cannot be altered by the derived class.

THE COSTS OF INHERITANCE

- Inheritance decreases the execution speed due to the increased time and effort it takes, the program to jump through all the levels of overloaded classes.
- Inheritance makes the two classes (base and inherited class) get tightly coupled. This means one cannot be used independently of each other.
- The changes made in the parent class will affect the behavior of child class too.
- The overuse of inheritance makes the program more complex.

ACCESS CONTROL(MEMBER ACCESS)

In Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element.

Types of Access Modifiers in Java

There are four types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

1. Default Access Modifier

- When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default.
- The default modifier is accessible only within package.
- It cannot be accessed from outside the package.
- It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A
{
        void msg()
        {
                System.out.println("Hello");
        }
}

//save by B.java
package mypack;
import pack.*;
class B
{
         public static void main(String args[])
        {
                A obj = new A();    //Compile Time Error
                obj.msg();          //Compile Time Error
        }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

2. private

- The private access modifier is accessible only within the class.
- The private access modifier is specified using the keyword **private**.
- The methods or data members declared as private are accessible only **within the class** in which they are declared.
- Any other **class of** the **same package will not be able to access** these members.
- Top-level classes or interfaces can not be declared as private because private means “only visible within the enclosing class”.

Example

- In this example, we have created two classes A and Simple.
- A class contains private data member and private method.
- We are accessing these private members from outside the class, so there is a compile-time error.

```
class A
{
        private int data=40;
        private void msg()
        {
                System.out.println("Hello java");}
        }

public class Simple
{
        public static void main(String args[])
        {
                A obj=new A();
                System.out.println(obj.data);        //Compile Time Error
                obj.msg();                           //Compile Time Error
  }
}
```

3. protected

- The protected access modifier is accessible within package and outside the package but through inheritance only.
- The protected access modifier is specified using the keyword **protected**.

Example

- In this example, we have created the two packages pack and mypack.
- The A class of pack package is public, so can be accessed from outside the package.
- But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A
{
        protected void msg()
        {
                System.out.println("Hello");
        }
}

//save by B.java
package mypack;
import pack.*;
class B extends A
{
        public static void main(String args[])
        {
                B obj = new B();
                obj.msg();
        }
}
```

4. public

- The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.
- The public access modifier is specified using the keyword **public**.

Example

```
//save by A.java
package pack;
public class A
{
	public void msg()
	{
		System.out.println("Hello");
	}
}
//save by B.java
package mypack;
import pack.*;
class B
{
	public static void main(String args[])
	{
		A obj = new A();
		 obj.msg();
	}
}
```

Table: class member access

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	YES	NO	NO	NO
Default	YES	YES	NO	NO
Protected	YES	YES	YES	NO
Public	YES	YES	YES	YES

SUPER KEYWORD

Super keyword in java is a reference variable that is used to refer parent class features.

Usage of Java super Keyword

1. Super keyword At Variable Level
2. Super keyword At Method Level
3. Super keyword At Constructor Level

- Whenever the derived class is inherits the base class features, there is a possibility that base class features are similar to derived class features and JVM gets an ambiguity.
- In order to differentiate between base class features and derived class features must be preceded by super keyword.

Syntax

super.baseclass features

1. Super Keyword at Variable Level

- Whenever the derived class inherit base class data members there is a possibility that base class data member are similar to derived class data member and JVM gets an ambiguity.
- In order to differentiate between the data member of base class and derived class, in the context of derived class the base class data members must be preceded by super keyword.

Syntax

super.baseclass datamember name

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 Keyword at Method Level

- The **super keyword** can also be used to invoke or call parent class method.
- It should be use in case of method overriding. In other word **super keyword** use when base class method name and derived class method name have same name.

Example

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

3. Super keyword At Constructor Level

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

```
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();
	}
}
```

FINAL KEWWORD

- It is used to make a variable as a constant, Restrict method overriding, Restrict inheritance.
- Final keyword is used to make a variable as a constant.
- This is similar to const in other language.

In java language final keyword can be used in following ways:

1. Final Keyword at Variable Level
2. Final Keyword at Method Level
3. Final Keyword at Class Level

1.Final at variable level

- A variable declared with the final keyword cannot be modified by the program after initialization.
- This is useful to universal constants, such as "PI".

Example

```
class Bike
{
        final int speedlimit=90;
        void run()
        {
                speedlimit=400;
        }
        public static void main(String args[])
        {
                Bike9 obj=new  Bike9();
                obj.run();
        }
}
```

Output: Compile Time Error

2.Final Keyword at method level

- It makes a method final, meaning that sub classes can not override this method. The compiler checks and gives an error if you try to override the method.
- When we want to restrict overriding, then make a method as a final.

Example

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

Output: It gives an error

3.Final Keyword at Class Level

It makes a class final, meaning that the class cannot be inheriting by other classes. When we want to restrict inheritance then make class as a final.

Example

```
        final class Bike
        {
        }
        class Honda1 extends Bike
        {
                void run()
                {
                        System.out.println("running safely with 100kmph");
                }
                public static void main(String args[])
                {
                        Honda1 honda= new Honda1();
                        honda.run();
                }
        }
```

Output: Compile Time Error

POLYMORPHISM

- The polymorphism is the process of defining same method with different implementation. That means creating multiple methods with different behaviors.

Types of Java polymorphism

The Java polymorphism is mainly divided into two types:

1. Compile-time Polymorphism(Method Overloading)
2. Runtime Polymorphism(Method Overriding)

Ad Hoc Polymorphism(Method Overloading)

Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as **method overloading**.

Example

```
class Addition
{
        void sum(int a, int b)
        {
                System.out.println(a+b);
        }
        void sum(int a, int b, int c)
        {
                System.out.println(a+b+c);
        }
        void sum(float a, float b)
        {
                System.out.println(a+b);
        }
 }
class Methodload
{
        public static void main(String args[])
        {
                Addition obj=new Addition();
                obj.sum(10, 20);
                obj.sum(10, 20, 30);
                obj.sum(10.05f, 15.20f);
         }
}
```

Pure Polymorphism(Method Overriding)

- Whenever same method name is existing in both base class and derived class with same types of parameters or same order of parameters is known as **method Overriding**.
- In a java programming language, pure polymorphism carried out with a method overriding concept.

Note: Without Inheritance method overriding is not possible.

Example

```
class Walking
{
    void walk()
    {
        System.out.println("Man walking fastly");
    }
}
class Man extends Walking
{
    void walk()
    {
        System.out.println("Man walking slowly");
        super.walk();
    }
}
class OverridingDemo
{
    public static void main(String args[])
    {
        Man obj = new Man();
        obj.walk();
    }
}
```

Note:

- Whenever we are calling overridden method using derived class object reference the highest priority is given to current class (derived class). We can see in the above example high priority is derived class.
- super. (super dot) can be used to call base class overridden method in the derived class.

ABSTRACT CLASS

- 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).
- An abstract class must be declared with an abstract keyword.
- 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.

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

There are two ways to achieve abstraction in java.

1. Abstract class (0 to 100%)
2. Interface (100%)

Syntax

```
abstract class className
{
    ……
}
```

ABSTRACT METHOD

- An abstract method contains only declaration or prototype but it never contains body or definition.
- In order to make any undefined method as abstract whose declaration is must be predefined by abstract keyword.

Syntax

```
abstract returntype methodName(List of formal parameter);
```

Example

```
abstract class Shape
{
        abstract void draw();
}
class Rectangle extends Shape
{
        void draw()
        {
                System.out.println("drawing rectangle");
        }
}
class Circle1 extends Shape
{
        void draw()
        {
                System.out.println("drawing circle");
        }
}
class TestAbstraction1
{
         static void main(String args[])
        {
                Shape s=new Circle1();
                s.draw();
        }
}
```

Example2

```
import java.util.*;
abstract class Shape
{
    int length, breadth, radius;
    Scanner input = new Scanner(System.in);
    abstract void printArea();
}
class Rectangle extends Shape
{
    void printArea()
    {
        System.out.println("*** Finding the Area of Rectangle ***");
        System.out.print("Enter length and breadth: ");
        length = input.nextInt();
        breadth = input.nextInt();
        System.out.println("The area of Rectangle is: " + length * breadth);
    }
}
class Triangle extends Shape
{
    void printArea()
    {
        System.out.println("\n*** Finding the Area of Triangle ***");
        System.out.print("Enter Base And Height: ");
        length = input.nextInt();
        breadth = input.nextInt();
        System.out.println("The area of Triangle is: " + (length * breadth) / 2);
    }
}
class Cricle extends Shape
{
    void printArea()
    {
        System.out.println("\n*** Finding the Area of Cricle ***");
        System.out.print("Enter Radius: ");
        radius = input.nextInt();
        System.out.println("The area of Cricle is: " + 3.14f * radius * radius);
    }
}
public class AbstractClassExample
{
    public static void main(String[] args)
    {
        Rectangle rec = new Rectangle();
        rec.printArea();
        Triangle tri = new Triangle();
        tri.printArea();
        Cricle cri = new Cricle();
        cri.printArea();
    }
}
```

OBJECT CLASS

- In java, the Object class is the super most class of any class hierarchy. The Object class in the java programming language is present inside the java.lang package.
- Every class in the java programming language is a subclass of Object class by default.
- The Object class is useful when you want to refer to any object whose type you don't know. Because it is the superclass of all other classes in java, it can refer to any type of object.

Method	Description	Return Value
getClass()	Returns Class class object	object
hashCode()	returns the hashcode number for object being used.	int
equals(Object obj)	compares the argument object to calling object.	boolean
clone()	Compares two strings, ignoring case	int
concat(String)	Creates copy of invoking object	object
toString()	returns the string representation of invoking object.	String
notify()	wakes up a thread, waiting on invoking object's monitor.	void
notifyAll()	wakes up all the threads, waiting on invoking object's monitor.	void
wait()	causes the current thread to wait, until another thread notifies.	void
wait(long,int)	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies.	void
finalize()	It is invoked by the garbage collector before an object is being garbage collected.	void

PACKAGES IN JAVA

A package is a collection of similar types of classes, interfaces and sub-packages.

Types of packages

Package are classified into two type which are given below.

1. Predefined or built-in package
2. User defined package

1. Predefined or built-in package

These are the packages which are already designed by the Sun Microsystem and supply as a part of java API, every predefined package is collection of predefined classes, interfaces and sub-package.

Following are the list of predefined packages in java

- **java.lang** – This package provides the language basics.
- **java.util** – This packages provides classes and interfaces (API's) related to collection frame work, events, data structure and other utility classes such as date.
- **java.io** – This packages provides classes and interfaces for file operations, and other input and output operations.
- **java.awt** – This packages provides classes and interfaces to create GUI components in Java.
- **java.time** – The main API for dates, times, instants, and durations.

2. User defined package

- If any package is design by the user is known as user defined package.
- User defined package are those which are developed by java programmer and supply as a part of their project to deal with common requirement.

DEFINING A PACKAGE

Rules to create user defined package

- Package statement should be the first statement of any package program.
- Choose an appropriate class name or interface name and whose modifier must be public.
- Any package program can contain only one public class or only one public interface but it can contain any number of normal classes.
- Package program should not contain any main() method.
- Modifier of constructor of the class which is present in the package must be public. (This is not applicable in case of interface because interface have no constructor.)
- The modifier of method of class or interface which is present in the package must be public (This rule is optional in case of interface because interface methods by default public)
- Every package program should be save either with public class name or public Interface name

- If you omit the package statement, the class names are put into the default package, which has no name.

Syntax

```
package packagename;
```

Example

```
package mypack;
```

Compile package programs

For compilation of package program first we save program with public className.java and it compile using below syntax:

Syntax

```
javac -d . className.java
```

Explanation

- In above syntax **"-d"** is a specific tool which tells to java compiler create a separate folder for the given package in given path.
- When we give specific path then it create a new folder at that location and when we use . (dot) then it crate a folder at current working directory.

Note: Any package program can be compile but can not be execute or run. These program can be executed through user defined program which are importing package program.

Example of Package Program

Package program which is save with A.java and compile by javac -d . A.java.

```
package mypack;
public class A
{
        public void show()
        {
                System.out.println("Sum method");
        }
}
```

IMPORTING PACKAGES

- To import the java package into a class, we need to use the java import keyword which is used to access the package and its classes into the java program.
- Use import to access built-in and user-defined packages into your java source file to refer to a class in another package by directly using its name.

syntax:

```
import package.name.ClassName;    // To import a certain class only
import package.name.*              // To import the whole package
```

Example:

```
import java.util.Date;             // imports only Date class
import java.io.*;                  // imports everything inside java.io package
```

Example

```
import mypack.A;
public class Hello
{
    public static void main(String args[])
    {
        A a=new A();
        a.show();
        System.out.println("show() class A");
    }
}
```

CLASSPATH

CLASSPATH can be set by any of the following ways:

- CLASSPATH can be set permanently in the environment:
- In Windows, choose control panel
- System
- Advanced
- Environment Variables
- choose “System Variables” (for all the users) or “User Variables” (only the currently login user)
- choose “Edit” (if CLASSPATH already exists) or “New”
- Enter “CLASSPATH” as the variable name
- Enter the required directories and JAR files (separated by semicolons) as the value (e.g., “.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar”).
- Take note that you need to include the current working directory (denoted by ‘.’) in the CLASSPATH.

To check the current setting of the CLASSPATH, issue the following command:

- > SET CLASSPATH
- CLASSPATH can be set temporarily for that particular CMD shell session by issuing the following command:
- > SET CLASSPATH=.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar
- Instead of using the CLASSPATH environment variable, you can also use the command-line option -classpath or -cp of the javac and java commands, for example,

  > java –classpath c:\javaproject\classes com.abc.project1.subproject2.MyClass3

INTERFACES

- **Interface** is similar to class which is collection of public static final variables (constants) and abstract methods.
- The interface is a mechanism to achieve fully abstraction in java. There can be only abstract methods in the interface. It is used to achieve fully abstraction and multiple inheritance in Java.

Why do we use an Interface?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.
- Any class can extend only 1 class but can any class implement infinite number of interface.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?
- The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public and static.

DIFFERENCE BETWEEN CLASS AND INTERFACE

Class	Interface
The keyword used to create a class is "class"	The keyword used to create an interface is "interface"
A class can be instantiated i.e., objects of a class can be created.	An Interface cannot be instantiated i.e. objects cannot be created.
Classes do not support multiple inheritance.	The interface supports multiple inheritance.
It can be inherited from another class.	It cannot inherit a class.
It can be inherited by another class using the keyword 'extends'.	It can be inherited by a class by using the keyword 'implements' and it can be inherited by an interface using the keyword 'extends'.
It can contain constructors.	It cannot contain constructors.
It cannot contain abstract methods.	It contains abstract methods only.
Variables and methods in a class can be declared using any access specifier(public, private, default, protected).	All variables and methods in an interface are declared as public.
Variables in a class can be static, final, or neither.	All variables are static and final.

DEFINING INTERFACES

The **interface** keyword is used to declare an interface.

Syntax

```
interface interface_name
{
        declare constant fields
        declare methods that abstract
}
```

Example

```
interface A
 {
        public static final int a = 10;
        void display();
}
```

IMPLEMENTING INTERFACES

A class uses the **implements** keyword to implement an interface.

Example

```
interface A
 {
        public static final int a = 10;
        void display();
}
class B implements A
{
        public void display()
        {
                System.out.println("Hello");
        }
}
class InterfaceDemo
{
        public static void main (String[] args)
        {
                B obj= new B();
                obj.display();
                System.out.println(a);
   }
}
```

APPLYING INTERFACES

To understand the power of interfaces, let's look at a more practical example.

Example:

```
interface IntStack
{
    void push(int item);
    int pop();
}
class FixedStack implements IntStack
{
    private int stck[];
    private int top;
    FixedStack(int size)
    {
        stck = new int[size];
        top = -1;
    }
    public void push(int item)
    {
        if(top==stck.length-1)
          System.out.println("Stack is full.");
        else
          stck[++top] = item;
    }
    public int pop()
    {
        if(top ==-1)
        {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
        return stck[top--];
    }
}
class InterfaceTest
{
    public static void main(String args[])
    {
        FixedStack mystack1 = new FixedStack(5);
        FixedStack mystack2 = new FixedStack(8);
        for(int i=0; i<5; i++)
            mystack1.push(i);
        for(int i=0; i<8; i++)
            mystack2.push(i);
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());
        for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}
```

VARIABLES IN INTERFACE

- Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class.
- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.

Example

```
interface SharedConstants
{
        int NO = 0;
        int YES = 1;
        int MAYBE = 2;
        int LATER = 3;
        int NEVER = 4;
}
class Question implements SharedConstants
{
    BufferedReader br=new BufferedReader(new InputStreamreader(System.in));
    int ask()
    {
            System.out.println("would u like to have a cup of coffee?)
            String ans=br.readLine();
            if (ans= ="no")
            return NO;
            else if (ans=="yes")
            return YES;
            else if (ans=="notnow")
            return LATER;
            else
            return NEVER;
     }
}
class AskMe
{
        public static void main(String args[])
        {
            Question q = new Question();
            System.out.println(q.ask());
        }
}
```

EXTENDING INTERFACES

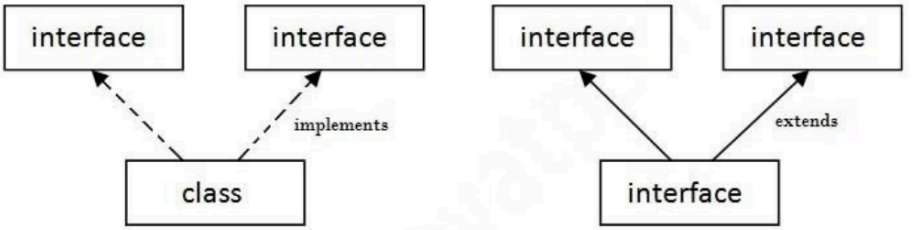
- One interface can inherit another by use of the keyword **extends**.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Example

```
interface A
{
    void meth1();
    void meth2();
}
interface B extends A
{
    void meth3();
}
class MyClass implements B
{
    public void meth1()
    {
        System.out.println("Implement meth1().");
    }
    public void meth2()
    {
        System.out.println("Implement meth2().");
    }
    public void meth3()
    {
        System.out.println("Implement meth3().");
    }
}
class InterfaceDemo
{
    public static void main(String args[])
    {
        MyClass ob = new MyClass();
        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}
```

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

Example

```
interface Printable
{
        void print();
}
interface Showable
{
        void show();
}
class A implements Printable,Showable
{
        public void print()
        {
                System.out.println("Hello");
        }
        public void show()
        {
                System.out.println("Welcome");
        }
        public static void main(String args[])
        {
                A obj = new A();
                obj.print();
                obj.show();
        }
}
```

STREAM BASED I/O (JAVA.IO)

- **Java I/O** (Input and Output) is used *to process the* input *and* produce the output.
- Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.
- We can perform **file handling in Java** by Java I/O API.

STREAM

In Java, streams are the sequence of data that are read from the source and written to the destination.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1. System.in: This is the **standard input stream** that is used to read characters from the keyboard or any other standard input device.

2. System.out: This is the **standard output stream** that is used to produce the result of a program on an output device like the computer screen.

3. System.err: This is the **standard error stream** that is used to output all the error data that a program might throw, on a computer screen or any standard output device.

TYPES OF STREAMS

Depending on the type of operations, streams can be divided into two primary classes:

- **InPutStream** – The InputStream is used to read data from a source.
- **OutPutStream** – The OutputStream is used for writing data to a destination.

Depending upon the data a stream can be classified into:

1. Byte Stream
2. Character Stream

1. BYTE STREAM

Java byte streams are used to perform input and output of 8-bit bytes.

Byte Stream Classes

All byte stream classes are derived from base abstract classes called **InputStream** and **OutputStream**.

InputStream Class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Subclasses of InputStream

In order to use the functionality of InputStream, we can use its subclasses. Some of them are:

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
DataInputStream	Contains method for reading java standard datatype
FileInputStream	Input stream that reads from a file

Methods of InputStream

The InputStream class provides different methods that are implemented by its subclasses. Here are some of the commonly used methods:

- **read() -** reads one byte of data from the input stream
- **read(byte[] array) -** reads bytes from the stream and stores in the specified array
- **available() -** returns the number of bytes available in the input stream
- **mark() -** marks the position in the input stream up to which data has been read
- **reset() -** returns the control to the point in the stream where the mark was set
- **close() -** closes the input stream

OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes.

Subclasses of OutputStream

In order to use the functionality of OutputStream, we can use its subclasses. Some of them are:

Stream class	Description
BufferedOutputStream	Used for Buffered Output Stream.
DataOutputStream	An output stream that contain method for writing java standard data type
FileOutputStream	Output stream that write to a file.
PrintStream	Output Stream that contain print() and println() method

Methods of OutputStream

The OutputStream class provides different methods that are implemented by its subclasses. Here are some of the methods:

- **write() -** writes the specified byte to the output stream
- **write(byte[] array) -** writes the bytes from the specified array to the output stream
- **flush() -** forces to write all data present in output stream to the destination
- **close() -** closes the output stream

2. CHARACTER STREAM

Character stream is used to read and write a single character of data.

Character Stream Classes

All the character stream classes are derived from base abstract classes **Reader** and **Writer.**

Reader Class

The Reader class of the java.io package is an abstract super class that represents a stream of characters.

Sub classes of Reader Class

In order to use the functionality of Reader, we can use its subclasses. Some of them are:

Stream class	Description
BufferedReader	Handles buffered input stream.
FileReader	Input stream that reads from file.
InputStreamReader	Input stream that translate byte to character

Methods of Reader

The Reader class provides different methods that are implemented by its subclasses. Here are some of the commonly used methods:

- **ready() -** checks if the reader is ready to be read
- **read(char[] array) -** reads the characters from the stream and stores in the specified array
- **read(char[] array, int start, int length) -** reads the number of characters equal to length from the stream and stores in the specified array starting from the start
- **mark() -** marks the position in the stream up to which data has been read
- **reset() -** returns the control to the point in the stream where the mark is set
- **skip() -** discards the specified number of characters from the stream

Writer Class

- The Writer class of the java.io package is an abstract super class that represents a stream of characters.
- Since Writer is an abstract class, it is not useful by itself. However, its subclasses can be used to write data.

Subclasses of Writer

Stream class	Description
BufferedWriter	Handles buffered output stream.
FileWriter	Output stream that writes to file.
PrintWriter	Output Stream that contain print() and println() method.

Methods of Writer

The Writer class provides different methods that are implemented by its subclasses. Here are some of the methods:

- **write(char[] array) -** writes the characters from the specified array to the output stream
- **write(String data) -** writes the specified string to the writer
- **append(char c) -** inserts the specified character to the current writer
- **flush() -** forces to write all the data present in the writer to the corresponding destination
- **close() -** closes the writer

READING CONSOLE INPUT

There are times when it is important for you to get input from users for execution of programs. To do this you need Java Reading Console Input Methods.

Java Reading Console Input Methods

1. Using BufferedReader Class
2. Using Scanner Class
3. Using Console Class

1. Using BufferedReader Class

- Reading input data using the BufferedReader class is the traditional technique. This way of the reading method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the console.
- The BufferedReader class has defined in the java.io package.
- We can use read() method in BufferedReader to read a character.

int read() throws IOException

Reading Console Input Characters Example:

```
import java.io.*;
class ReadingConsoleInputTest
{
        public static void main(String args[])
        {
            char ch;
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Enter characters, char 'x' to exit.");
            do
            {
                    ch = (char) br.read();
                    System.out.println(ch);
            } while(ch != 'x');
        }
}
```

How to read a string input in java?

readLine() method is used to read the string in the BufferedReader.

Program to take String input from Keyboard in Java

```
import java.io.*;
class MyInput
{
  public static void main(String[] args)
  {
    String text;
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    text = br.readLine();    //Reading String
    System.out.println(text);
  }
}
```

2. Using the Scanner Class

Scanner is one of the predefined class which is used for reading the data dynamically from the keyboard.

Import Scanner Class in Java

java.util.Scanner

Constructor of Scanner Class

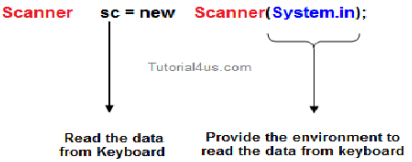
Scanner(InputStream)

This constructor create an object of Scanner class by talking an object of InputStream class. An object of InputStream class is called **in** which is created as a static data member in the System class.

Syntax of Scanner Class in Java

Scanner sc=new Scanner(System.in);

Here the object **'in'** is use the control of keyboard



Instance methods of Scanner Class

S.No	Method	Description
1	public byte nextByte()	Used for read byte value
2	public short nextShort()	Used for read short value
3	public int nextInt()	Used for read integer value
4	public long nextLong()	Used for read numeric value
5	public float nextLong()	Used for read numeric value
6	public double nextDouble()	Used for read double value
7	public char nextChar()	Used for read character
8	public boolean nextBoolean()	Used for read boolean value
9	public String nextLine()	Used for reading any kind of data in the form of String data.

Example of Scanner Class in Java

```
import java.util.Scanner
public class ScannerDemo
{
    public static void main(String args[])
    {
        Scanner  s=new  Scanner(System.in);
        System.out.println("Enter first no= ");
        int num1=s.nextInt();
        System.out.println("Enter second no= ");
        int num2=s.nextInt();
        System.out.println("Sum of no is= "+(num1+num2));
    }
}
```

3. Using the Console Class

- This is another way of reading user input from the console in Java.
- The Java Console class is be used to get input from console. It provides methods to read texts and passwords.
- If you read password using Console class, it will not be displayed to the user.
- The Console class is defined in the java.io class which needs to be imported before using the console class.

Example

```
import java.io.*;
class consoleEg
{
    public static void main(String args[])
    {
        String name;
        System.out.println ("Enter your name: ");
        Console c = System.console();
        name = c.readLine();
        System.out.println ("Your name is: " + name);
    }

}
```

WRITING CONSOLE OUTPUT

- print and println methods in System.out are mostly used for console output.
- These methods are defined by the class PrintStream which is the type of object referenced by System.out.
- System.out is the byte stream.
- PrintStream is the output derived from OutputStream. write method is also defined in PrintStream for console output.

void write(int byteval)

//Java code to Write a character in Console Output

```
import java.io.*;
class WriteCharacterTest
{
      public static void main(String args[])
      {
            int byteval;
            byteval = 'J';
            System.out.write(byteval);
            System.out.write('\n');
      }
}
```