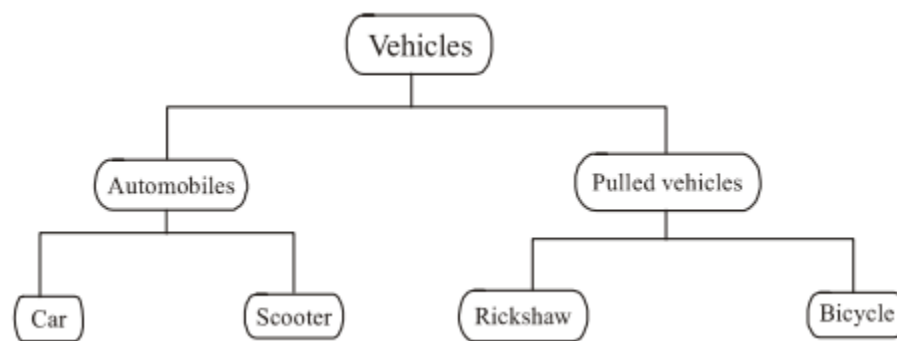


**UNIT II****INHERITANCE, PACKAGES AND INTERFACES****INHERITANCE:**

Inheritance can be defined as the process where an object of a class acquires characteristics from the object of another class.

Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.

The process by which one class acquires the properties(data members) and functionalities (methods) of another class is called inheritance.



The class, which is inherited by the other classes, is known as superclass or base class or parent class and the class, which inherits the properties of the base class, is called sub class or derived class or child class. The sub-class can further be inherited to form other derived classes.

**Terms used in Inheritance**

**Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

**Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

**Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

**Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

**The syntax of Java Inheritance**

```
class Subclass-name extends Superclass-name
```

```
{  
  
//methods and fields  
  
}
```

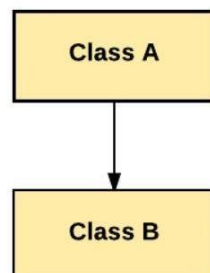
The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

## TYPES OF INHERITANCE

- Single level inheritance
- Multiple Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

### Single level inheritance

This type of inheritance occurs for only a single class. Only one class is derived from the parent class. In this type of inheritance, the properties are derived from a single parent class. As the properties are derived from only a single base class the reusability of a code is facilitated along with the addition of new features.



### Syntax of Single Inheritance

```
class base class  
{  
    .... methods  
}  
class derivedClass name extends baseClass  
{  
    methods ...  
}
```

### Example:

```
class Employee
{
    void salary()
    {
        System.out.println("Salary= 200000");
    }
}
class Programmer extends Employee    // Programmer class inherits from Employee class
{
    void bonus()
    {
        System.out.println("Bonus=50000");
    }
}
class single_inheritance
{
    public static void main(String args[])
    {

        Programmer p = new Programmer();
        p.salary(); // calls method of super class
        p.bonus(); // calls method of sub class
    }
}
```

**Output:**

Salary= 200000

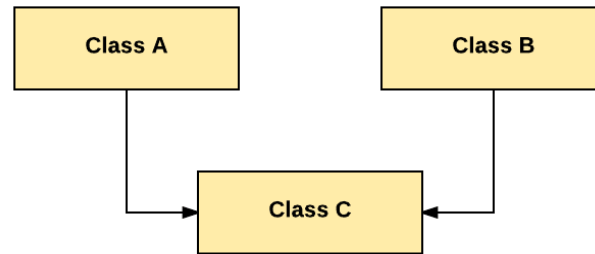
Bonus=50000

**Multiple Inheritance**

Multiple inheritances is a type of inheritance where a subclass can inherit features from more than one parent class.

In multiple inheritances the newly derived class can have more than one superclass.

Java does not support multiple inheritance. This means that we can not use multiple inheritance in java but a class can inherit properties from many classes in java using interfaces.

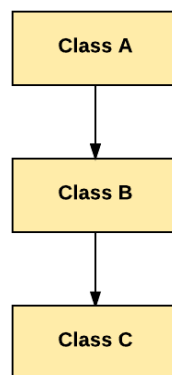


Class C is derived from the two classes Class A and Class B. In other words it can be described that subclass C inherits properties from both Class A and B.

### Multi-level Inheritance

In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.

In the multilevel inheritance in java, the inherited features are also from the multiple base classes as the newly derived class from the parent class becomes the base class for another newly derived class.

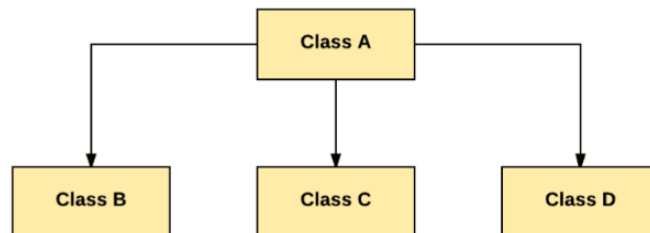


```
package com.techvidvan.inheritance;
//Base class
class Person
{
    public void show()
    {
        System.out.println("Student inheriting properties from Person");
    }
}
class Student extends Person
{
    public void show1()
    {
```

```
        System.out.println("I am a Student who belongs to Person class");
    }
}
//child class
class EngineeringStudent extends Student
{
    // defining additional properties to the child class
    public void show2()
    {
        System.out.println("Engineering Student inheriting properties from Student");
    }
}
public class MultilevelDemo
{
    public static void main(String args[])
    {
        EngineeringStudent obj = new EngineeringStudent();
        obj.show();
        obj.show1();
        obj.show2();
    }
}
```

### Hierarchical Inheritance

Hierarchical inheritance is a type of inheritance in Java where multiple derived classes inherit the properties of a parent class. It allows all the child classes to inherit methods and fields from their parent class.



### Syntax

```
Class P {
// fields and methods
}
```

```
class C1 extends P {
// fields and methods
}
```

```
}
```

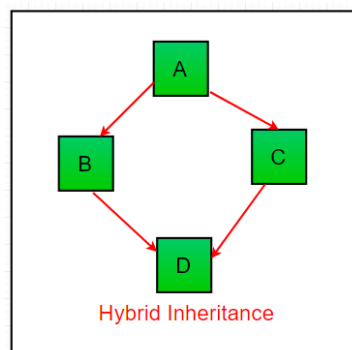
```
class C2 extends P {  
    // fields and methods  
}
```

```
class C3 extends P {  
    // fields and methods  
}
```

```
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  
    }}
```

## Hybrid Inheritance

Hybrid inheritance is a combination of more than two types of inheritances single and multiple. It can be achieved through interfaces only as multiple inheritance is not supported by Java. It is basically the combination of simple, multiple, hierarchical inheritances.



```
class C
```

```
{
    public void disp()
    {
        System.out.println("C");
    }
}

class A extends C
{
    public void disp()
    {
        System.out.println("A");
    }
}

class B extends C
{
    public void disp()
    {
        System.out.println("B");
    }
}

class D extends A
{
    public void disp()
    {
        System.out.println("D");
    }
    public static void main(String args[]){

        D obj = new D();
        obj.disp();
    }
}
```

**Output: D**

### **ADVANTAGES OF INHERITANCE:**

- **Code reusability** - public methods of base class can be reused in derived classes
- **Data hiding** – private data of base class cannot be altered by derived class
- **Overriding**- With inheritance, we will be able to override the methods of the base class in the derived class

### **USING SUPER**

- super keyword is similar to this keyword in Java.
  - It is used to refer to the immediate parent class of the object.
- Super keyword used in 3 levels:

It can be used to refer immediate parent class instance variable when both parent and child class have member with same name

It can be used to invoke immediate parent class method when child class has overridden that method.

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

### **Use of super with variables:**

#### **Example:**

```
class SuperCls
{
int x = 20;
}
class SubCls extends SuperCls
{
int x = 80;
void display()
{
System.out.println("Super Class x: " + super.x); //print x of super class
System.out.println("Sub Class x: " + x); //print x of subclass
}}
class Main
{
public static void main(String[] args)
{
SubCls obj = new SubCls();
obj.display();
}}
```

#### **Sample Output:**

```
Super Class x: 20
Sub Class x: 80
```

### **Use of super with methods:**

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 (Method Overriding).

#### **Example:**

```
class SuperCls
{
int x = 20;
void display()          //display() in super class
{
```



```
System.out.println("Super Class x: " + x);
}}
class SubCls extends SuperCls
{
int x = 80;
void display()           //display() redefined in sub class – method overriding
{
System.out.println("Sub Class x: " + x);
super.display(); // invoke super class display()
}}
class Main
{
public static void main(String[] args)
{
SubCls obj = new SubCls();
obj.display();
}}
```

**Sample Output:**

Sub Class x: 80

Super Class x: 20

**Use of super with constructors:**

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

**Syntax:**

```
super();
```

- super() if present, must always be the first statement executed inside a subclass constructor.
- When we invoke a super() statement from within a subclass constructor, we are invoking the immediate super class constructor

**Example:**

```
class Vehicle
{

Vehicle( )
{
System.out.println("Vehicle is created");
} }
class Bike extends Vehicle
```

```

{
Bike()
{
super();           //will invoke parent class constructor
System.out.println("Bike is created");
}}
class SuperClassConst
{
public static void main(String args[])
{
Bike b=new Bike();
}}

```

### THE FINAL KEYWORD IN JAVA

The final keyword in Java indicates that no further modification is possible. Final is a keyword in java used for restricting some functionalities. We can declare variables, methods and classes with final keyword.

Using final with inheritance

Final can be used with:

Class

Methods

Variables

#### Class declared as final:

When a class is declared as final, it cannot be extended further. Here is an example what happens within a program

```

final class stud {
    // Methods cannot be extended to its sub class
}
class books extends stud {
    void show() {
        System.out.println("Book-Class method");
    }
    public static void main(String args[]) {
        books B1 = new books();
        B1.show();
    }
}

```

```
error: cannot inherit from final stud
class books extends stud{
^
```

### Method declared as final

A method declared as final cannot be overridden; this means even when a child class can call the final method of parent class without any issues, but the overriding will not be possible. Here is a sample program showing what is not valid within a Java program when a method is declared as final.

#### Example

```
class stud {
    final void show() {
        System.out.println("Class - stud : method defined");
    }
}
class books extends stud {
    void show() {
        System.out.println("Class - books : method defined");
    }
    public static void main(String args[]) {
        books B2 = new books();
        B2.show();
    }
}
```

### Variable declared as final

Once a variable is assigned with the keyword final, it always contains the same exact value. Again things may happen like this; if a final variable holds a reference to an object then the state of the object can be altered if programmers perform certain operations on those objects, but the variable will always refer to the same object. A final variable that is not initialized at the time of declaration is known as a blank final variable. If you are

```
import java.util.*;
import java.lang.*;
import java.io.*;
```

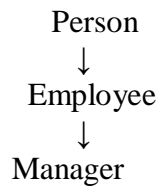
initialize the blank final variable within the constructor  
tion error.

```
class stud {
    final int val;
    stud() {
        val = 60;
    }
    void method() {
        System.out.println(val);
    }
    public static void main(String args[]) {
        stud S1 = new stud();
        S1.method();
    }
}
```

## CREATING MULTILEVEL HIERARCHY

### How to create a multilevel hierarchy in Java

In simple inheritance, a subclass or derived class derives the properties from its parent class, but in multilevel inheritance, a subclass is derived from a derived class. One class inherits the only single class. Therefore, in multilevel inheritance, every time ladder increases by one. The lowermost class will have the properties of all the **superclass**.



#### Example:

```
class A
{
A()
{
System.out.println("A constructor");
}
void Amethod( )
{
System.out.println("method of A");
}}
class B extends A
{
B( )
{
System.out.println("B constructor");
}
void Bmethod( )
{
System.out.println("method of B");
}
void welcome( )
{
System.out.println("class B-welcome method");
}}
class C extends B
```

```
{
C( )
{
System.out.println("C constructor");
}
void welcome()
{
System.out.println("class C- welcome method");
}
public static void main(String args[])
{
C oc=new C( );
oc.Amethod( );
oc.Bmethod( );
oc.welcome( );
}}
```

**Output:**

A constructor  
B constructor  
C constructor  
method of A  
method of B  
method of A

**FORMS OF INHERITANCE:**

All objects eventually inherit from Object, which provides useful methods such as equals and toString.

Inheritance gets used for a number of purposes in typical object-oriented programming:

**specialization** -- the subclass is a special case of the parent class

**specification** -- the superclass just specifies which methods should be available but doesn't give code. This is supported in java by interfaces and abstract methods.

**construction** -- the superclass is just used to provide behavior, but instances of the subclass don't really act like the superclass.

**extension** -- subclass adds new methods, and perhaps redefines inherited ones as well.

**limitation** -- the subclass restricts the inherited behavior.

**combination** -- multiple inheritance. Provided in part by implementing multiple interfaces.

### **FORMS OF INHERITANCE (- INHERITANCE FOR SPECIALIZATION -)**

Most commonly used inheritance and sub classification is for specialization. Always creates a subtype, and the principles of substitutability is explicitly upheld. It is the most ideal form of inheritance.

An example of subclassification for specialization is; public class PinBallGame extends Frame {

// body of class

}

#### **Specialization**

1. By far the most common form of inheritance is for specialization.
  - Child class is a specialized form of parent class
  - Principle of substitutability holds
2. A good example is the Java hierarchy of Graphical components in the AWT:
  - Component
    - Label
    - Button
    - TextComponent
  - TextArea
  - TextField
    - CheckBox
    - ScrollBar

### **FORMS OF INHERITANCE (- INHERITANCE FOR SPECIFICATION -)**

This is another most common use of inheritance. Two different mechanisms are provided by Java, interface and abstract, to make use of subclassification for specification. Subtype is formed and substitutability is explicitly upheld.

Mostly, not used for refinement of its parent class, but instead is used for definitions of the properties provided by its parent.

class FireButtonListener implements ActionListener

{

// body of class

```
}  
  
class B extends A {  
  
// class A is defined as abstract specification class  
  
}
```

### **Specification**

The next most common form of inheritance involves specification. The parent class specifies some behavior, but does not implement the behavior

- Child class implements the behavior
- Similar to Java interface or abstract class
- When parent class does not implement actual behavior but merely defines the behavior that will be implemented in child classes

### **Example, Java 1.1 Event Listeners:**

ActionListener, MouseListener, and so on specify behavior, but must be subclassed.

### **FORMS OF INHERITANCE (- INHERITANCE FOR CONSTRUCTION -)**

Child class inherits most of its functionality from parent, but may change the name or parameters of methods inherited from parent class to form its interface.

This type of inheritance is also widely used for code reuse purposes. It simplifies the construction of newly formed abstraction but is not a form of subtype, and often violates substitutability.

Example is Stack class defined in Java libraries.

### **Construction**

The parent class is used only for its behavior, the child class has no is-a relationship to the parent.

- Child modify the arguments or names of methods

An example might be subclassing the idea of a Set from an existing List class.

- Child class is not a more specialized form of parent class; no substitutability

**FORMS OF INHERITANCE (- INHERITANCE FOR EXTENSION -)**

Subclassification for extension occurs when a child class only adds new behavior to the parent class and does not modify or alter any of the inherited attributes.

Such subclasses are always subtypes, and substitutability can be used.

Example of this type of inheritance is done in the definition of the class Properties which is an extension of the class Hashtable.

**Generalization or Extension**

The child class generalizes or extends the parent class by providing more functionality

- ✓ In some sense, opposite of subclassing for specialization

The child doesn't change anything inherited from the parent, it simply adds new features

- ✓ Often used when we cannot modify existing base parent class

Example, ColoredWindow inheriting from Window

- ✓ Add additional data fields
- ✓ Override window display methods

**FORMS OF INHERITANCE (- INHERITANCE FOR LIMITATION -)**

Subclassification for limitation occurs when the behavior of the subclass is smaller or more restrictive than the behavior of its parent class.

Like subclassification for extension, this form of inheritance occurs most frequently when a programmer is building on a base of existing classes.

Is not a subtype, and substitutability is not proper.

**LIMITATION**

- The child class limits some of the behavior of the parent class.
- Example, you have an existing List data type, and you want a Stack
- Inherit from List, but override the methods that allow access to elements other than top so as to produce errors.

**FORMS OF INHERITANCE (- INHERITANCE FOR COMBINATION -)**

This type of inheritance is known as multiple inheritance in Object Oriented Programming.



Although the Java does not permit a subclass to be formed by inheritance from more than one parent class, several approximations to the concept are possible.

Example of this type is Hole class defined as;

```
class Hole extends Ball implements PinBallTarget{  
  
    // body of class  
  
}
```

### **Combination**

- Two or more classes that seem to be related, but its not clear who should be the parent and who should be the child.
- Example: Mouse and TouchPad and JoyStick
- Better solution, abstract out common parts to new parent class, and use subclassing for specialization.

### **THE BENEFITS OF INHERITANCE**

- Software Reusability (among projects)
- Increased Reliability (resulting from reuse and sharing of well-tested code)
- Code Sharing (within a project)
- Consistency of Interface (among related objects)
- Software Components
- Rapid Prototyping (quickly assemble from pre-existing components)
- Polymorphism and Frameworks (high-level reusable components)
- Information Hiding

### **THE COSTS OF INHERITANCE**

- ❖ Execution Speed
- ❖ Program Size
- ❖ Message-Passing Overhead
- ❖ Program Complexity (in overuse of inheritance)

### **HIERARCHICAL ABSTRACTIONS:**

Hierarchical abstraction in Java refers to the concept of organizing code and data in a hierarchical structure, allowing for the creation of a hierarchy of classes and objects. This helps in representing real-world relationships and structures more effectively. In Java, hierarchical abstraction is achieved through the use of classes and inheritance.

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

**Ways to achieve Abstraction:**

There are two ways to achieve abstraction in java

- **Abstract class**
- **Interface**

### **Abstract classes and Java Abstract methods**

- An abstract class is a class that is declared with an abstract keyword.
- An abstract method is a method that is declared without implementation.
- An abstract class may or may not have all abstract methods. Some of them can be concrete methods
- A method-defined abstract must always be redefined in the subclass, thus making overriding compulsory or making the subclass itself abstract.
- Any class that contains one or more abstract methods must also be declared with an abstract keyword.
- There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the new operator.
- An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

// Abstract class representing a shape

```
abstract class Shape
{
    // Abstract method for calculating area (no implementation here)
    abstract double calculateArea();
}
```

// Concrete class representing a circle

```
class Circle extends Shape {
    double radius;
    Circle(double radius) {
        this.radius = radius;
    }

    // Implementation of the abstract method for calculating the area of a circle
    @Override
    double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Creating a Circle object
        Circle myCircle = new Circle(5.0);

        // Using the abstracted method to calculate the area
        double area = myCircle.calculateArea();
        System.out.println("Area of the circle: " + area);
    }
}
```

```
}
```

hierarchical abstraction is achieved through the use of classes and inheritance.

Hierarchical abstraction in Java refers to the concept of organizing code and data in a hierarchical structure, allowing for the creation of a hierarchy of classes and objects. This helps in representing real-world relationships and structures more effectively. In Java, hierarchical abstraction is achieved through the use of classes and inheritance.

Here's a basic explanation of hierarchical abstraction in Java:

### **i) Classes and Objects:**

In Java, everything is an object, and objects are instances of classes.

A class is a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of the class will have.

Objects represent real-world entities, and classes group similar objects together.

### **ii) Inheritance:**

Inheritance is a key feature of object-oriented programming that allows a class to inherit properties and behaviors from another class.

A class that is inherited is called a superclass or parent class, and the class that inherits from it is called a subclass or child class.

The subclass can reuse and extend the functionality of the superclass.

### **iii) Hierarchical Inheritance:**

Hierarchical inheritance is a type of inheritance where a single superclass is inherited by multiple subclasses.

```
// Superclass
class Animal
{
    void eat() {
        System.out.println("This animal eats food.");
    }
}
```

```
// Subclass 1
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}
```

```
// Subclass 2
class Cat extends Animal {
    void meow() {
        System.out.println("The cat meows.");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat(); // Inherited from Animal
    }
}
```

```
myDog.bark(); // Specific to Dog
```

```
Cat myCat = new Cat();  
myCat.eat(); // Inherited from Animal  
myCat.meow(); // Specific to Cat  
}}
```

### **BASE CLASS OBJECT:**

The term "base class" is commonly referred to as the superclass or parent class. An object in Java is an instance of a class.

Object Class in Java is the topmost class among all the classes in Java. We can also say that the Object class in Java is the parent class for all the classes. It means that all the classes in Java are derived classes and their base class is the Object class.

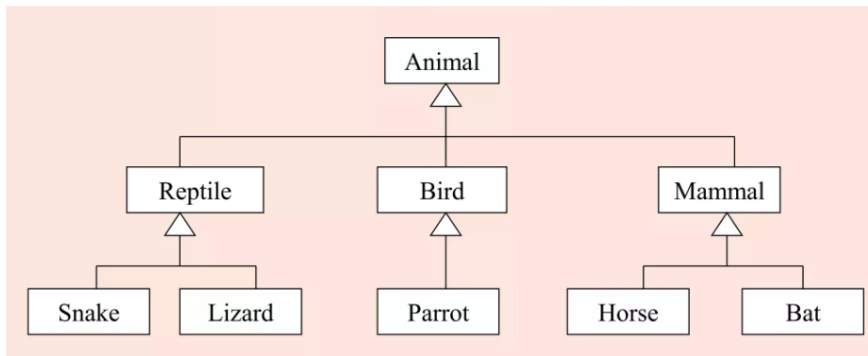
// Base class or superclass

```
class Animal {  
    String name;  
  
    // Constructor  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    // Method  
    public void makeSound() {  
        System.out.println("Some generic sound");  
    }  
}
```

// Subclass or derived class

```
class Dog extends Animal {  
    // Additional attributes and methods specific to Dog can be added here  
  
    // Constructor  
    public Dog(String name) {  
        // Call the constructor of the superclass (Animal)  
        super(name);  
    }  
  
    // Override the makeSound method  
    @Override  
    public void makeSound() {  
        System.out.println("Woof! Woof!");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        // Creating an object of the base class (Animal)  
        Animal genericAnimal = new Animal("Generic Animal");  
        genericAnimal.makeSound(); // Output: Some generic sound  
    }  
}
```

```
// Creating an object of the derived class (Dog)
Dog myDog = new Dog("Buddy");
myDog.makeSound(); // Output: Woof! Woof!
}}
```



### SUBCLASS:

A subclass (or derived class) is a class that inherits properties and behaviors from another class, known as the superclass or base class.

The subclass can add new attributes or methods, and it can also override or extend the functionality of the inherited methods.

In Java, the extends keyword is used to create a subclass.

```
// Base class
class Animal {
    // ...
}
// Subclass
class Dog extends Animal {
    // ...
}
```

### SUBTYPE:

- The term “subtype” is used to describe the relationship between different types that follow the principle of “substitution”.
- If we consider two types(classes or interfaces) A and B, type B is called as a subtype of A, if the following two conditions are satisfied:
  - 1) The instance(object) of type B can be legally assigned to the variable of type A.
  - 2) The instance(object) of type B can be used by the variable of type A without any observable change in its behavior.

```
Animal myAnimal = new Dog();           // Dog is a subtype of Animal
```

**SUBSTITUTABILITY:**

- Substitutability means, the type of the variable does not have to match with the type of the value assigned to that variable.
- Substitutability cannot be achieved in conventional languages in C, but can be achieved in Object Oriented languages like Java.
- We have already seen the concept of “Assigning a subclass object to superclass variable or reference”. This is called substitutability. Here I am substituting the superclass object with the object of subclass.
- When new subclasses are constructed by extending the superclass, we can say that substitutability is satisfied under the following conditions:
  - 1) Instances of the subclass must contain all the fields(instance variables) of the super class.
  - 2) Instances of the subclass must implement, through inheritance all functionality defined for the super class.
  - 3) Thus, the instance of a subclass will contain all the characteristics and behavior of the super class. So, the object of the subclass can be substituted in the place of a superclass object.

### Example

```
abstract class Shape
{
    int dim1;
    int dim2;
    Shape(int x, int y)
    {
        dim1 = x; dim2 = y;
    }
    abstract void area();
    abstract void volume();
}

class Triangle extends Shape
{
    Triangle(int x, int y)
    {
        super(x,y);
    }
    void area()
    {
        System.out.println("Area of triangle is: "+(dim1*dim2)/2);
    }
    void volume()
    {
        System.out.println("Volume of the triangle");
    }
}
```

```

class Rectangle extends Shape
{
    Rectangle(int x, int y)
    {
        super(x,y);
    }
    void area()
    {
        System.out.println("Area of rectangle is: "+(dim1*dim2));
    }
    void volume()
    {
        System.out.println("Volume of the rectangle");
    }
}

class ShapeDemo
{
    public static void main(String args[])
    {
        Shape s;
        s = new Triangle(10,20);
        s.area();
        s.volume();
        s = new Rectangle(20,40);
        s.area();
        s.volume();
    }
}

```

## MEMBER ACCESS RULES IN JAVA

The member access rules determines whether a sub class can use a property of it's super class or it can only access or it can neither access nor access.

Member access rules dictate how classes, fields, methods, and nested classes can be accessed or inherited by other classes.

- ✓ At the top level: public or package-private
- ✓ At the member level: public, private, protected

A class may be declared with the 'public' modifier, in that case that class is visible to all classes everywhere.

At the member level, there are three different access modifiers are there: 'private', 'protected' and 'public'.

**private :** If private access modifier is applied to an instance variable, method or with a constructor in side a class then they will be accessed inside that class only not out side of the class.

**Example:**

```

class A
{
private int x=10;
} class B extends A
{
int y=20;
System.out.println(x); //Illegal access to x ;
}

```

If you make any class constructor private, you can't create the instance/object of that class from outside the class. for example:

```

class A
{
    int x;
    private A(int k) // private constructor
    {
        x=k;
    }
}
class Test
{
    public static void main(String args[])
    {
        A ob=new A(10); //Compile time error
    }
}

```

**protected:** If protected access modifier is applied to an instance variable, method or with a constructor in side a class then they will be accessed inside the package only in which class is present and, in addition, by a sub class in another package.

```

class MyClass {

    protected int protectedField;

    protected void protectedMethod() {

        // Code here

    }
}

```

**public:** The class, variable, method or a constructor with public access modifier can be accessed from anywhere.

```

public class MyClass {

    public int publicField;
}

```



```
public void publicMethod() {  
  
    // Code here  
  
}}
```

**POLYMORPHISM:**

The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism is a person who at the same time can have different characteristics.

A man at the same time is a father, a husband, and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

```
class Polygon {  
  
    // method to render a shape  
    public void render() {  
        System.out.println("Rendering Polygon...");  
    }  
}  
  
class Square extends Polygon {  
  
    // renders Square  
    public void render() {  
        System.out.println("Rendering Square...");  
    }  
}  
  
class Circle extends Polygon {  
  
    // renders circle  
    public void render() {  
        System.out.println("Rendering Circle...");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
  
        // create an object of Square  
        Square s1 = new Square();  
        s1.render();  
  
        // create an object of Circle  
        Circle c1 = new Circle();  
        c1.render(); } }
```

**Types of Polymorphism:**

- Compile-time Polymorphism (Static Binding or Method Overloading)
- Runtime Polymorphism (Dynamic Binding or Method Overriding)

**Method Overloading**

Method overloading is a concept that allows to declare multiple methods with same name but different parameters in the same class.

Java supports method overloading and always occur in the same class(unlike method overriding).

```
void func() { ... }
```

```
void func(int a) { ... }
```

```
float func(double a) { ... }
```

```
float func(int a, float b) { ... }
```

The func() method is overloaded. These methods have the same name but accept different arguments.

Method overloading is one of the ways through which java supports polymorphism.

Polymorphism is a concept of object oriented programming that deal with multiple forms.

There are two different ways of method overloading.

1. Different datatype of arguments
2. Different number of arguments

**Method overloading by changing data type of arguments.****Example:**

In this example, we have two sum() methods that take integer and float type arguments respectively.

```
class Calculate
{
    void sum (int a, int b)
    {
        System.out.println("sum is" +(a+b)) ;
    }
    void sum (float a, float b)
    {
        System.out.println("sum is" +(a+b));
    }
    Public static void main (String[] args)
    {
        Calculate cal = new Calculate();
        cal.sum (8,5);    //sum(int a, int b) is method is called.
        cal.sum (4.6f, 3.8f); //sum(float a, float b) is called.
    }
}
```

**Output:**

Sum is 13

Sum is 8.4

### **Method overloading by changing no. of argument.**

#### **Example:**

```
class Demo
{
    void multiply(int l, int b)
    {
        System.out.println("Result is" +(l*b)) ;
    }
    void multiply(int l, int b,int h)
    {
        System.out.println("Result is" +(l*b*h));
    }
    public static void main(String[] args)
    {
        Demo ar = new Demo();
        ar.multiply(8,5); //multiply(int l, int b) is method is called
        ar.multiply(4,6,2); //multiply(int l, int b,int h) is called
    }
}
```

#### **Output:**

Result is 40

Result is 48

### **METHOD OVERRIDING:**

Method overriding is a process of overriding base class method by derived class method with more specific definition.

Method overriding performs only if two classes have is-a relationship. It mean class must have inheritance. In other words, It is performed between two classes using inheritance relation.

In overriding, method of both class must have same name and equal number of parameters.

Method overriding is also referred to as runtime polymorphism because calling method is decided by JVM during runtime.

The key benefit of overriding is the ability to define method that's specific to a particular subclass type.

### **Rules for Method Overriding**

1. Method name must be same for both parent and child classes.
2. Access modifier of child method must not restrictive than parent class method.
3. Private, final and static methods cannot be overridden.

```

class Animal
{
    public void eat()
    {
        System.out.println("Eat all eatables");
    }
}

class Dog extends Animal
{
    public void eat() //eat() method overridden by Dog class.
    {
        System.out.println("Dog like to eat meat");
    }
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();
    }
}

```

**Output:**

Dog like to eat meat

**DIFFERENCE BETWEEN METHOD OVERLOADING AND METHOD OVERRIDING**

<b>METHOD OVERLOADING</b>	<b>METHOD OVERRIDING</b>
Parameter must be different and name must be same.	Both name and parameter must be same.
Compile time polymorphism.	Runtime polymorphism.
Access specifier can be changed.	Access specifier cannot be more restrictive than original method(can be less restrictive).
Increase readability of code.	Increase reusability of code.
It is performed within a class	It is performed between two classes using inheritance relation.
It should have methods with the same name but a different signature.	It should have methods with same name and signature.
It cannot have the same return type.	It should always have the same return type.
It can be performed using the static method	It cannot be performed using the static method
It uses static binding	It uses the dynamic binding.
Access modifiers and Non-access modifiers can be changed.	Access modifiers and Non-access modifiers cannot be changed.
Private, static, final methods can be overloaded	Private, static, final methods cannot be overloaded

## ABSTRACT CLASSES

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body). It needs to be extended and its method implemented.

### Syntax:

```
abstract class classname
```

```
{
}
```

## ABSTRACT METHOD

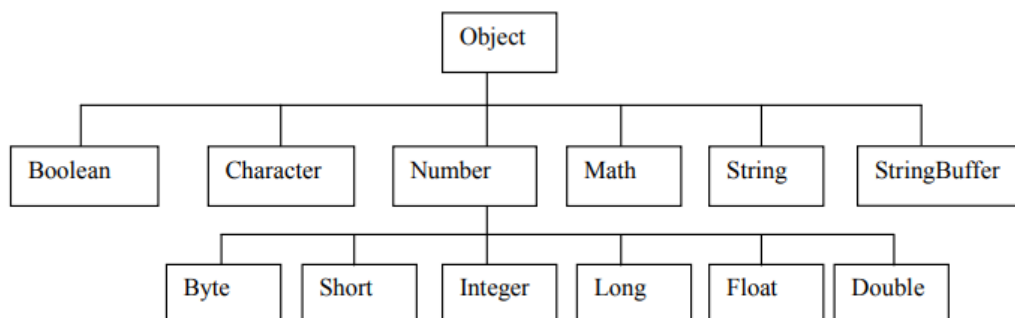
A method that is declared as abstract and does not have implementation is known as abstract method. The method body will be defined by its subclass.

### Syntax:

```
abstract returntype functionname ();           //No definition
```

## THE OBJECT CLASS

The Object class is the parent class of all the classes in java by default (directly or indirectly). The java.lang.Object class is the root of the class hierarchy. Some of the Object class are Boolean, Math, Number, String etc.



Object class Methods	Description
boolean equals(Object)	Returns true if two references point to the same object.
String toString()	Converts object to String
void notify() void notifyAll() void wait()	Used in synchronizing threads
void finalize()	Called just before an object is garbage collected
Object clone()	Returns a new object that are exactly the same as the current object
int hashCode()	Returns a hash code value for the object.

## PACKAGES

### INTRODUCTION TO PACKAGES

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

Thus, package is a container of a group of related classes where some of the classes are accessible and are exposed and others are kept for internal purpose. We can reuse existing classes from the packages as many time as we need it in our program. Package names and directory structure are closely related. For example if a package name is college.staff.csc, then there are three directories, college, staff and csc such that csc is present in staff and staff is present college.

#### Packages are used for:

**Re-usability:** The classes contained in the packages of another program can be easily reused

**Name Conflicts:** Packages help us to uniquely identify a class, for example, we can have two classes with the name Employee in two different packages, company.sales.Employee and company.marketing.Employee.

**Controlled Access:** Offers access protection such as protected classes, default classes and private class. Protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.

**Data Encapsulation:** They provide a way to hide classes, preventing other programs from accessing classes that are meant for internal use only

**Maintenance:** With packages, you can organize your project better and easily locate related classes

#### Types of packages in Java:

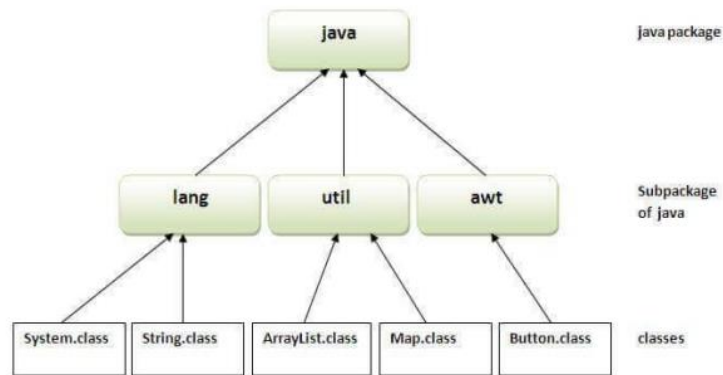
- 1) **User defined package:** The package we create is called user-defined package.
- 2) **Built-in package:** The already defined package like java.io.\*, java.lang.\* etc are known as built-in packages.

#### Built-in Packages:

Built-in packages or predefined packages are those that come along as a part of JDK (Java Development Kit) to simplify the task of Java programmer. They consist of a huge number of predefined classes and interfaces that are a part of Java API's. Some of the commonly used built-in packages are:

**JAVA API Packages:**

Java API (Application Program Interface) provides a large numbers of classes grouped into different packages according to functionality. Most of the time we use the packages available with the Java API



**java.lang:** Contains language support classes (e.g. `Class` which defines primitive data types, math operations). This package is automatically imported.

**java.io:** Contains classes for supporting input / output operations.

**java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support for Date / Time operations.

**java.applet:** Contains classes for creating Applets.

**java.awt:** Contain classes for implementing the components for graphical user interfaces (like button, menus etc).

**java.net:** Contain classes for supporting networking operations.

**User Defined Packages:** User-defined packages are those which are developed by users in order to group related classes, interfaces and sub packages

**CREATING PACKAGES:**

To create our own package, we must first declare the name of package using the `package` keyword followed by a package name. This must be first statement in a java source file.

```
Package firstpackage;
```

```
Public class FirstClass
```

```
{
```

.....

..... ( Body of class )

}

Here package name is firstPackage.

The class FirstClass is now considered a part of this package.

This file is saved as FirstClass.java and located in a directory named firstPackage.

When the source file is compiled , java will create a .class file and store it in the same directory.

**Example:**

```
package mypackage;  
public class student  
{  
Statement;  
}
```

**Example:**

```
Package mypack;  
public class Simple  
{  
public static void main(String args[])  
{  
System.out.println("Welcome to package");  
} } }
```

**How to compile Java packages:**

This is just like compiling a normal java program. If you are not using any IDE, you need to follow the steps given below to successfully compile your packages:

1. java -d directory javafilename

**Example:**

```
javac -d . Simple.java
```

How to run java package program:

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

**To create our own package following steps are there:**



1. Declare the package at the beginning of a file using the form `package packagename;`
2. Define the class that is to be put in the package and declare it public.
3. Create a subdirectory under the directory where the main source files are stored.
4. Store the programs as `classname.java` file in the subdirectory created.
5. Compile the file. This creates `.class` file in subdirectory. Remember that name of the package must be same as the directory under which this file is saved.

### Accessing a Package

#### How to access package from another package:

There are three ways to access the package from outside the package.

1. `import package.*;`
2. `import package.classname;`
3. fully qualified name.

#### 1. **import package.\*;**

If you use `package.*` then all the classes and interfaces of this package will be accessible but not sub packages. The `import` keyword is used to make the classes and interface of another package accessible to the current package.

```
import packagename.*;
```

Here the package name may denote a single package or a hierarchy of packages. The star (\*) indicates the entire package hierarchy.

```
import firstPackage.secondPackage.MyClass ;
```

By the use of this statement ,all the members of the class `MyClass` can be directly accessed using the class name .

```
import package.ClassA ;  
class PackageTest  
{  
Public static void main(String args[ ])  
{  
ClassA obj = new ClassA( ) ;  
Obj.displayA( ) ;  
}}
```

**Example of package that import the packagename.\*:**

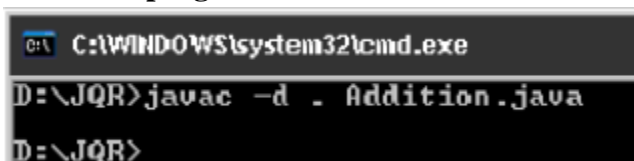
```
//save by A.java
package pack;
public class A
{
public void msg()
{
System.out.println("Hello java");
}
}
//save by B.java
package mypack;
import pack.*;
class B
{
public static void main(String args[])
{
A obj = new A();
obj.msg();
}}
```

**Output:** Hello java

**Program 1: Write a program to create a package pack with Addition class.**

```
//creating a package
package pack;
public class Addition
{
private double d1,d2;
public Addition(double a,double b)
{
d1 = a;
d2 = b;
}
public void sum()
{
System.out.println ("Sum of two given numbers is : " + (d1+d2) );
}
}
```

**Compiling the above program:**



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac -d . Addition.java
D:\JQR>
```

**2) Using packagename.classname:**

If you import package.classname then only declared class of this package will be accessible.

**Example of package by import package.classname:**

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

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

**Output: Hello**

**3) Using fully qualified name:**

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

**Example of package by import fully qualified name:**

```
//save by A.java
package pack;
public class A
{
    public void msg()
```

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

**Output: Hello**

### SUB PACKAGES IN JAVA

A package inside another package is known as sub package.

```
package letmecalculate.multiply;
public class Multiplication
{
int product(int a, int b)
{
return a*b;
}}
```

If I create a package inside letmecalculate package then that will be called sub package.

Lets say I have created another package inside letmecalculate and the sub package name is multiply.

### CLASSPATH:

The CLASSPATH is an environment variable that tells the Java compiler where to look for class files to import.

If the package pack is available in different directory, in that case the compiler should be given information regarding the package location by mentioning the directory name of the package in the 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

### **Creating our first package:**

**File name – ClassOne.java**

```
package package_name;
public class ClassOne
{
public void methodClassOne()
{
System.out.println("Hello there its ClassOne");
}}
```

### **Creating our second package:**

**File name – ClassTwo.java**

```
package package_one;
public class ClassTwo
{
public void methodClassTwo()
{
System.out.println("Hello there i am ClassTwo");
}}
```

## **ACCESS PROTECTION IN JAVA PACKAGES**

In java, the access modifiers define the accessibility of the class and its members.

In java, the package is a container of classes, sub-classes, interfaces, and sub-packages. The class acts as a container of data and methods. So, the access modifier decides the accessibility of class members across the different packages.

In java, the accessibility of the members of a class or interface depends on its access specifiers. The following table provides information about the visibility of both data members and methods.

Java has four access modifiers, and they are default, private, protected, and public.

**Public:** Same class, Same package, Subclasses, Everyone.

Public members can be accessed from everywhere within your application either through inheritance or through object reference.

**Private:** Same class.

Private members are accessible only within the same class and also are not inherited.

**Protected:** Same class, Same package, Subclasses.

- ✓ members of the same class
- ✓ members of any class in the same package (same as default)
- ✓ subclasses (any level of subclass hierarchy) in other packages (only through inheritance).
- ✓ object reference (Parent or Child) in subclasses

**Default:** Same class, Same package.

- ✓ Default (no modifier) members can be accessed by members of the same class and members of any class in the same package.
- ✓ Default members are inherited by another class only if both parent and child are in same package.

## INTERFACES:

- The interface keyword is used to declare an interface.
- An interface is just like Java Class, but it only has static constants and abstract method.
- Java uses Interface to implement multiple inheritance.
- A Java class can implement multiple Java Interfaces.
- All methods in an interface are implicitly public and abstract.

## Declaring Interfaces:

### Syntax

```
Interface
{
//methods
}
```

### Example:

```
interface Animal
{
public void eat();
public void travel();
}
```

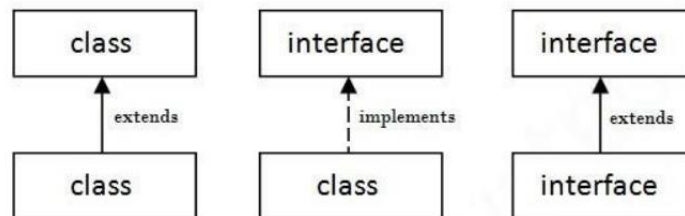
## Implementing Interfaces:

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

- To use an interface in your class, append the keyword "implements" after your class name followed by the interface name.

**Example for Implementing Interface:**

```
public class MammalInt implements Animal
{ }
```

**Extending Interfaces**


An interface can extend another interface, similarly to the 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 Program on Extending Interface:**

```
import java.io.*;
interface student
{
    public void store(int a,String b);
    public void display();
}
interface marks extends student
{
    public void read(int a,int b,int c);
    public void compute();
}
class Demo5 implements marks
{
    int rno;
    String name;
    int m1,m2,m3;
    public void store(int a,String b)
    {
        rno=a;
        name=b;
    }
    public void display()
```

```
{
System.out.println("The Student Roll Number is "+rno);
System.out.println("The Student Name is "+name);
}
public void read(int x, int y, int z)
{
m1=x;
m2=y;
m3=z;
}
public void compute()
{
int tot=m1+m2+m3;
float avg=(tot)/3;
System.out.println("The total is "+tot);
System.out.println("The average is "+avg);
}
public static void main(String args[])
{
Demo5 d=new Demo5();
d.store(01,"Aman");
d.display();
d.read(40,50,65);
d.compute();
}}
```

**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.

**public interface Hockey extends Sports, Event**  
if the Hockey interface extended both Sports and Event



**Nested Interfaces:**

In java, an interface may be defined inside another interface, and also inside a class. The interface that defined inside another interface or a class is known as nested interface. The nested interface is also referred as inner interface.

The nested interface cannot be accessed directly. We can only access the nested interface by using outer interface or outer class name followed by dot( . ), followed by the nested interface name.

- The nested interface declared within an interface is public by default.
- The nested interface declared within a class can be with any access modifier.
- Every nested interface is static by default.

```
interface OuterInterface{
    void outerMethod();

    interface InnerInterface{
        void innerMethod();
    }
}

class OnlyOuter implements OuterInterface{
    public void outerMethod() {
        System.out.println("This is OuterInterface method");
    }
}

class OnlyInner implements OuterInterface.InnerInterface{
    public void innerMethod() {
        System.out.println("This is InnerInterface method");
    }
}

public class NestedInterfaceExample {

    public static void main(String[] args) {
        OnlyOuter obj_1 = new OnlyOuter();
        OnlyInner obj_2 = new OnlyInner();

        obj_1.outerMethod();
        obj_2.innerMethod();
    }
}
```

```
}
```

**Variables in Interfaces:**

We 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. When we include that interface in a class (that is, when you “implement” the interface), all of those variable names will be in scope as constants.

If an interface contains no methods, then any class that includes such an interface doesn’t actually implement anything. It is as if that class were importing the constant fields into the class name space as final variables.

**Example: Java program to demonstrate variables in interface.**

```
interface left
{
int i=10;
}
interface right
{
int i=100;
}
class Test implements left,right
{
public static void main(String args[])
{
System.out.println(left.i);
System.out.println(right.i);
}}
```

**Example : Java program to implement interface and inheriting the properties from a class.**

```
interface Teacher
{
void display1();
}
class Student
{
void display2()
{
System.out.println("Hi I am Student");
}}
class College extends Student implements Teacher
{
public void display1()
```

```

{
System.out.println("Hi I am Teacher");
}}
class Interface_Class
{
public static void main(String args[])
{
College c=new College();
c.display1();
c.display2();
}}

```

### APPLYING INTERFACE:

To understand the power of interfaces, let's look at a more practical example. In earlier chapters, you developed a class called Stack that implemented a simple fixed-size stack. However, there are many ways to implement a stack.

For example, the stack can be of a fixed size or it can be "growable." The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods **push()** and **pop()** define the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let's look at two examples. First, here is the interface that defines an integer stack. Put this in a file called IntStack.java.

**This interface will be used by both stack implementations.**

```

// Define an integer stack interface.
interface IntStack
{
void push(int item); // store an item
int pop();           // retrieve an item
}

```

### Applications are:

- Abstractions
- Multiple Inheritance

### Difference between Interface and Abstract class:

Abstract Class	Interface
Contains some abstract methods and some concrete methods	Only abstract methods
Contains instance variables	Only static and final variables
Doesn't support multiple inheritance	support multiple inheritance
public class Apple extends Food { ... }	public class Person implements Student,Athlete,Chef { ... }

## EXPLORING JAVA.IO

### Input/output

- Java I/O is a powerful concept, which provides the all input and output operations. Most of the classes of I/O streams are available in java.io package.
- Java has many input and output streams that are used to read and write data.

### Stream

- Java performs I/O through Streams.
- A stream means continuous flow of data.
- In Java, stream is basically a sequence of bytes, which has a continuous flow between Java programs and data storage.

### Types of Stream

Stream is basically divided into following types based on data flow direction.

#### Input Stream:

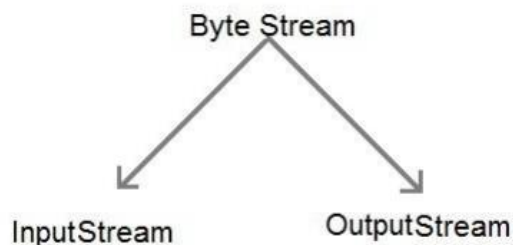
Input stream is represented as an input source. It is used to read the binary data from the source.

#### Output Stream:

Output stream represent a destination source. It is basically used to send out/write the data to destination.

#### Byte Streams:

Byte stream is used to input and output to perform 8-bits bytes.



### SOME IMPORTANT BYTE STREAM CLASSES.

STREAM CLASS	DESCRIPTION
Buffered Input Stream	Used for Buffered Input Stream.
Buffered Output Stream	Used for Buffered Output Stream.
Data Input Stream	Contains method for reading java standard data type
Data Output Stream	An output stream that contain method for writing java standard data type
File Input Stream	Input stream that reads from a file
File Output Stream	Output stream that write to a file.
Input Stream	Abstract class that describe stream input.
Output Stream	Abstract class that describe stream output.
PrintStream	Output Stream that contain print() and println() method

### Byte Stream Classes are in divided in two groups

**Input Stream Classes** - These classes are subclasses of an abstract class, Input Stream and they are used to read bytes from a source (file, memory or console).

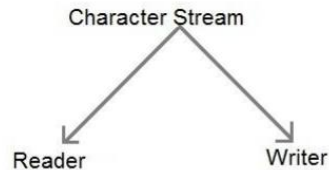
Input Stream is an abstract class and hence we can't create its object but we can use its subclasses for reading bytes from the input stream.

**Output Stream Classes** - These classes are subclasses of an abstract class, Output Stream and they are used to write bytes to a destination (file, memory or console).

Output Stream class is a base class of all the classes that are used to write bytes to a file, memory or console. Output Stream is an abstract class and hence we can't create its object but we can use its subclasses for writing bytes to the output stream.

### Character Streams:

Character stream basically works on 16 bit-Unicode value convention. This stream is used to read and write data in the format of 16 bit Unicode characters.



### SOME IMPORTANT CHARACTER STREAM CLASSES:

STREAM CLASS	DESCRIPTION
Buffered Reader	Handles buffered input stream.
Buffered Writer	Handles buffered output stream.
File Reader	Input stream that reads from file.
File Writer	Output stream that writes to file
Input Stream Reader	Input stream that translate byte to character
Output Stream Reader	Output stream that translate character to byte
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output
Print Writer	Output Stream that contain print() and println() method.

### READING CONSOLE INPUT

We use the object of Buffered Reader class to take inputs from the keyboard.

To take input from a user, we use Buffered Reader class by creating an object of it. For that, we have to write the following code.

```
BufferedReader b = new BufferedReader(new InputStreamReader(System.in));
```

**BufferedReader** - This is a class that is used for taking character input.

**b** - object of BufferedReader class

**InputStreamReader** - It converts bytes to characters.

**System.in** - It is input stream. User inputs are read from this.

Thus, we are taking user input from System.in which is converted from bytes to characters by the class InputStreamReader. This value is stored in the object b of the class BufferedReader.

### Reading data

Once we have taken input from the user, we need to read the data. Let's see how to read data.

### Reading characters

To read characters, read() method is used with the object of the BufferedReader class. Since read function returns an integer value, we need to convert it to character by typecasting it. The version of read( ) that we will be using is

**int read( ) throws IOException**

There are many ways to read data from the console input keyboard. For example:

- InputStreamReader
- Console
- Scanner
- DataInputStream etc

**InputStreamReader class:** InputStreamReader class can be used to read data from keyboard. It performs two tasks:

1. connects to input stream of keyboard
2. converts the byte-oriented stream into character-oriented stream

**BufferedReader class:** BufferedReader class can be used to read data line by line by readLine() method

In below example, we are connecting the BufferedReader stream with the InputStreamReader stream for reading the line by line data from the keyboard.

```
import java.io.*;
class IODemo
{
    public static void main(String args[])throws Exception
    {
        InputStreamReader r=new InputStreamReader(System.in);
        BufferedReader br=new BufferedReader(r);
        System.out.println("Enter your name");
        String name=br.readLine();
        System.out.println("Welcome "+name);
    } }
```

### Example: Reading characters from the keyboard

```
class Test
{
    public static void main( String args[])
    {
        BufferedReader b = new Bufferedreader(new InputstreamReader(System.in));
```

```
char ch = (char)b.read();  
}}
```

```
import java.io.*;  
class BRRead  
{  
public static void main(String args[]) throws IOException  
{  
char c;  
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
System.out.println("Enter characters, 'q' to quit.");  
// read characters  
do  
{  
c = (char) br.read();  
System.out.println(c);  
}  
while(c != 'q');  
}}
```

**Scanner Class:**

Java Scanner class is part of the java.util package. It was introduced in Java 1.5 release. It is used to read input data from different sources like input streams, users, files, etc.

METHOD	DESCRIPTION
public String next()	returns the next token from the scanner
public String nextLine()	moves the scanner position to the next line and returns the value as a string.
public byte nextByte()	scans the next token as a byte.
public short nextShort()	scans the next token as a short value.
public int nextInt()	scans the next token as an int value
public long nextLong()	scans the next token as a long value.
public float nextFloat()	scans the next token as a float value.
public double nextDouble()	scans the next token as a double value

**Example of the Scanner class**

```
import java.util.Scanner;  
class ScannerTest  
{  
public static void main(String args[])  
{  
Scanner sc=new Scanner(System.in);  
System.out.println("Enter your rollno");
```

```
int rollno=sc.nextInt();
System.out.println("Enter your name");
String name=sc.next();
System.out.println("Enter your fee");
double fee=sc.nextDouble();
System.out.println("Rollno:"+rollno+" name:"+name+" fee:"+fee);
}}
```

### **Reading strings**

We use `readLine()` method with the object of the `BufferedReader` class. Its general form is shown here:

#### **String readLine( ) throws IOException**

```
class Test
{
public static void main( String args[])
{
BufferedReader b = new BufferedReader(new InputStreamReader(System.in));
String s = b.readLine();
}}
```

#### **// Read a string from console using a BufferedReader**

```
import java.io.*;
class BRReadLines
{
public static void main(String args[]) throws IOException
{
// create a BufferedReader using System.in
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String str;
System.out.println("Enter lines of text.");
System.out.println("Enter 'stop' to quit.");
do
{
str = br.readLine();
System.out.println(str);
}
while(!str.equals("stop"));
}}
```



## WRITING CONSOLE OUTPUT

In java, there are two methods to write console output.

- Using print() and println() methods
- Using write() method

### 1. Writing console output using print() and println() methods

The PrintStream is a built-in class that provides two methods print() and println() to write console output. The print() and println() methods are the most widely used methods for console output.

Both print() and println() methods are used with System.out stream.

The print() method writes console output in the same line. This method can be used with console output only.

The println() method writes console output in a separate line (new line). This method can be used with console and also with other output sources.

#### Example

```
public class WritingDemo
{
    public static void main(String[] args)
    {
        int[] list = new int[5];
        for(int i = 0; i < 5; i++)
            list[i] = i*10;
        for(int i:list)
            System.out.print(i);    //prints in same line
        System.out.println("");
        for(int i:list)
            System.out.println(i); //Prints in separate lines
    }
}
```

### 2. Writing console output using write() method

Alternatively, the PrintStream class provides a method write() to write console output.

The write() method takes integer as argument, and writes its ASCII equivalent character on to the console, it also accepts escape sequences.

#### Example

```
public class WritingDemo
{
    public static void main(String[] args)
    {
```

```
int[] list = new int[26];
for(int i = 0; i < 26; i++)
{
list[i] = i + 65;
}
for(int i:list)
{
System.out.write(i);
System.out.write('\n');
}}
```