Java arrays

- Normally, an array is a collection of similar type of elements which has contiguous memory location.
- **Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.
- Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.
- Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.
- In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces.
- We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

- **Instantiation of an Array in Java**

arrayRefVar=**new** datatype[size];

Eg.

Int [] a=new int[size];

Example program

```
class Testarray{
    public static void main(String args[]){
    int a[]=new int[5];//declaration and instantiation
            a[0]=10;//initialization
            a[1]=20;
            a[2]=70;
            a[3]=40;
            a[4]=50;
    //traversing array
    for(int i=0;i<a.length;i++)//length is the property of array
            System.out.println(a[i]);
}
}
```

declaration, instantiation and initialization

int a[]={33,3,4,5};//declaration, instantiation and initialization

For-each Loop for Java Array

- We can also print the Java array using **for-each loop**.
- The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

```
for(data_type variable: array){
//body of the loop
}
```

//Java Program to print the array elements using for-each loop

```
class Testarray1{
    public static void main(String args[]){
    int arr[]={33,3,4,5};
    //printing array using for-each loop
    for(int i:arr)
    System.out.println(i);
  }
 }
```

Passing array to the method

```
//Java Program to demonstrate the way of passing an array
//to method.
class Testarray2{
//creating a method which receives an array as a parameter
    static void min(int arr[]){
    int min=arr[0];
    for(int i=1;i<arr.length;i++)
     if(min>arr[i])
      min=arr[i];

    System.out.println(min);
}
    public static void main(String args[]){
        int a[]={33,3,4,5};//declaring and initializing an array
        min(a);//passing array to method
}
}
```

Anonymous Array in Java

```
//Java Program to demonstrate the way of passing an anony
    mous array
//to method.
public class TestAnonymousArray{
    //creating a method which receives an array as a parameter
    static void printArray(int arr[]){
        for(int i=0;i<arr.length;i++)
          System.out.println(arr[i]);
}
public static void main(String args[]){
    printArray(new int[]{10,22,44,66});//passing anonymous
    array to method
}
}
```

Returning from array

```
//Java Program to return an array from the method
class TestReturnArray{
//creating method which returns an array
static int[] get(){
   return new int[]{10,30,50,90,60};
}
    public static void main(String args[]){
        //calling method which returns an array
        int arr[]=get();
        //printing the values of an array
        for(int i=0;i<arr.length;i++)
        System.out.println(arr[i]);
}
}
```

ArrayIndexOutOfBoundsException

- The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

```
//Java Program to demonstrate the case of
//ArrayIndexOutOfBoundsException in a Java Array.
public class TestArrayException{
    public static void main(String args[]){
                int arr[]={50,60,70,80};
                for(int i=0;i<=arr.length;i++){
                        System.out.println(arr[i]);
                }
        }
}
```

Multi Dimensional Array

- Dig out yourself!!!!!!!

ArrayList

- The ArrayList class is a resizable array, which can be found in the java.util package.
- The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one).
- While elements can be added and removed from an ArrayList whenever you want.
- The syntax is also slightly different:

```
import java.util.ArrayList; // import the ArrayList class
ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

Methods on ArrayList object

Add Items

- The ArrayList class has many useful methods. For example, to add elements to the ArrayList, use the add() method:

Q.Write a java program to add items in the ArrayList object.

Access an Item

- To access an element in the ArrayList, use the get() method and refer to the index number:

Eg: cars.get(0);

Change an Item

- To modify an element, use the set() method and refer to the index number:

Eg: cars.set(0, “lamborgini");

Remove an Item

- To remove an element, use the remove() method and refer to the index number:

Eg: cars.remove(0);

To remove all the elements in the ArrayList, use the clear() method:

Eg: cars.clear();

ArrayList Size

- To find out how many elements an ArrayList have, use the size method:

Eg: cars.size();

Loop Through an ArrayList

- Loop through the elements of an ArrayList with a for loop, and use the size() method to specify how many times the loop should run:

Eg:

```
for (int i = 0; i < cars.size(); i++)
  {
    System.out.println(cars.get(i));
  }
```

You can also loop through an ArrayList with the **for-each** loop:

```
for (String i : cars)
{
  System.out.println(i);
}
```

- Remember that a String in Java is an object (not a primitive type).
- To use other types, such as int, you must specify an equivalent wrapper class: Integer.
- For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc:

Sort an ArrayList

- Another useful class in the java.util package is the Collections class, which include the sort() method for sorting lists alphabetically or numerically:

```
ArrayList<Integer> myNumbers = new ArrayList<Integer>();
  myNumbers.add(33);
  myNumbers.add(15);
  myNumbers.add(20);
  myNumbers.add(34);
  myNumbers.add(8);
  myNumbers.add(12);
Collections.sort(myNumbers);
```

Object oriented basic features of Java

- Class and Object,
- Overloading,
- Access Privileges,
- Interface,
- Inner Class,
- Final and Static Modifiers,
- Packages,
- Inheritance,
- Overriding.

Class and Object

- Everything in Java is associated with classes and objects, along with its attributes and methods. For example:
- A Class is like an object constructor, or a "blueprint" for creating objects.
- An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.
- in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

Object contd…..

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:**

  An object identity is typically implemented via a unique ID.

  The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

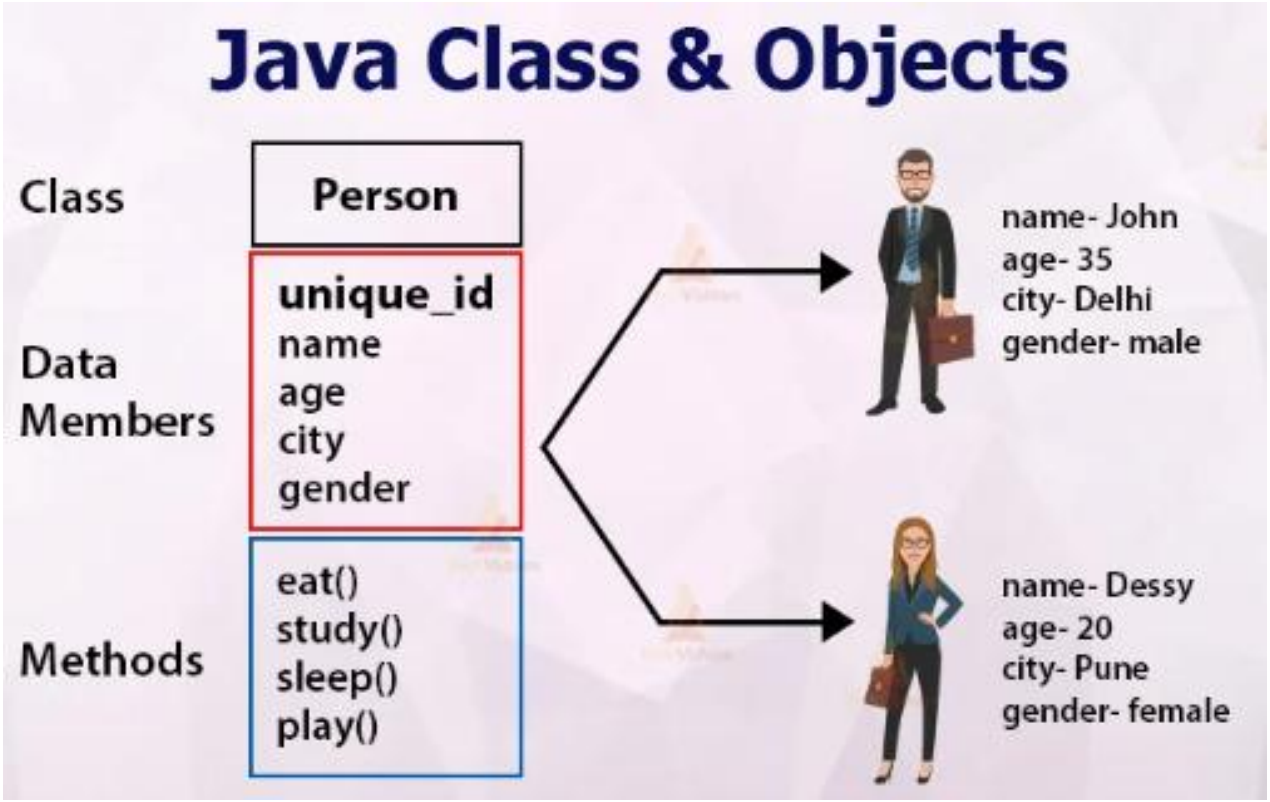
Object contd…

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

- **An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.


Java Class & Objects
Class
Person
Data Members
unique_id
name
age
city
gender
Methods
eat()
study()
sleep()
play()
name- John
age- 35
city- Delhi
gender- male
name- Dessy
age- 20
city- Pune
gender- female

Class in Java

- A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
- A class in Java can contain:
- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**

Syntax to declare class

```
class <class_name>{
    field;
    method;
}
```

Instance variable in Java

- A variable which is created inside the class but outside the method is known as an instance variable.
- Instance variable doesn't get memory at compile time.
- It gets memory at runtime when an object or instance is created.
- That is why it is known as an instance variable.

Method in Java

- In Java, a method is like a function which is used to expose the behavior of an object.
  - Advantage of Method
  - Code Reusability
  - Code Optimization

new keyword in Java

- The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

3 Ways to initialize object

- There are 3 ways to initialize object in Java.
- By reference variable
- By method
- By constructor

1) Object and Class Example: Initialization through reference

- Initializing an object means storing data into the object.

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a whit
     e space
 }
}
```

We can also create multiple objects and store information in it through reference variable

```
class Student{
 int id;
 String name;
}
class TestStudent3{
	public static void main(String args[]){
		//Creating objects
		Student s1=new Student();
		Student s2=new Student();
		//Initializing objects
		s1.id=101;
		s1.name="Sonoo";
		s2.id=102;
		s2.name="Amit";
		//Printing data
		System.out.println(s1.id+" "+s1.name);
		System.out.println(s2.id+" "+s2.name);
	}
}
```

2) Object and Class Example: Initialization through method

```
class Student{
    int rollno;
    String name;
 void insertRecord(int r, String n){
     rollno=r;
     name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
    class TestStudent4{
     public static void main(String args[]){
         Student s1=new Student();
         Student s2=new Student();
         s1.insertRecord(111,"Karan");
         s2.insertRecord(222,"Aryan");
         s1.displayInformation();
         s2.displayInformation();
     }
}
```

3) Object and Class Example: Initialization through a constructor

```
class Employee{
        int id;
        String name;
        float salary;
   Employee(int i, String n, float s) {
      id=i;
      name=n;
      salary=s;
   }
   void display(){System.out.println(id+" "+name+" "+salary);}
}
public class TestEmployee {
public static void main(String[] args) {
   Employee e1=new Employee(101,"ajeet",45000);
   Employee e2=new Employee(102,"irfan",25000);
   Employee e3=new Employee(103,"nakul",55000);
   e1.display();
   e2.display();
   e3.display();
}
}
```

Object and Class Example: main within the class

```
//Java Program to illustrate how to define a class and fields

//Defining a Student class.
class Student{
    //defining fields
    int id;//field or data member or instance variable
    String name;
    //creating main method inside the Student class
    public static void main(String args[]){
        //Creating an object or instance
        Student s1=new Student();//creating an object of Student
        //Printing values of the object
        System.out.println(s1.id);//accessing member through reference variable
        System.out.println(s1.name);
    }
}
```

Object and Class Example: main outside the class

```
//Java Program to demonstrate having the main method in

//another class
//Creating Student class.
class Student{
    int id;
    String name;
   }
   //Creating another class TestStudent1 which contains the main method
   class TestStudent1{
        public static void main(String args[]){
            Student s1=new Student();
            System.out.println(s1.id);
            System.out.println(s1.name);
        }
}
```

Example : Calculate area of rectangle

```
class Rectangle{
      int length;
      int width;
      void insert(int l, int w){
          length=l;
          width=w;
      }
      void calculateArea(){System.out.println(length*width);}
     }
     class TestRectangle1{
           public static void main(String args[]){
            Rectangle r1=new Rectangle();
            Rectangle r2=new Rectangle();
            r1.insert(11,5);
            r2.insert(3,15);
            r1.calculateArea();
            r2.calculateArea();
           }
}
```

Overloading

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.
- Method overloading *increases the readability of the program*.

Different ways to overload the method

- There are two ways to overload the method in java
  - By changing number of arguments
  - By changing the data type

1) Method Overloading: changing no. of arguments

```
class Addition{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
    System.out.println(Adder.add(20,30));
    System.out.println(Adder.add(40,50,60));
}}
```

2) Method Overloading: changing data type of arguments

```
class Addition{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a
   +b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(30,40));
System.out.println(Adder.add(15.5,11.7));
}}
```

Overloading Constructors

In addition to overloading normal methods, one can also overload constructors.

```
class Box {
      double width;
      double height;
      double depth;
     // constructor used when all dimensions specified
      Box(double w, double h, double d)
       {
              width = w;
              height = h;
              depth = d; }
      // constructor used when no dimensions specified
      Box() {
         width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
      }
     // constructor used when cube is created
      Box(double len) {
              width = height = depth = len; } // compute and return volume
      double volume() {
              return width * height * depth;  } }
```

```
class OverloadCons
{
 public static void main(String args[])
 {
// create boxes using the various constructors Box mybox1 = new Box(10, 20,
    15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);
    double vol; // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol); // get volume of second
    box
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol); // get volume of cube
     vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
}
}
```

Static Variables and Methods

- There will be times when you will want to define a class member that will be used independently of any object of that class.
- Normally a class member must be accessed only in conjunction with an object of its class.
- However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword static.
- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be static.
- The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist.
- Instance variables declared as static are, essentially, global variables.
- When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

Methods declared as static have several restrictions:

- They can only call other static methods.
- They must only access static data.
- They cannot refer to this or super in any way. (The keyword super relates to inheritance.)

If you need to do computation in order to initialize your static variables, you can declare a static block which gets executed exactly once, when the class is first loaded.

Final Variables

A variable can be declared as final. Doing so prevents its contents from being modified.

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

(In this usage, final is similar to const in C/C++/C#.)

For example:

```
final int FILE_NEW = 1;

final int FILE_OPEN = 2;

final int FILE_SAVE = 3;

final int FILE_SAVEAS = 4;

final int FILE_QUIT = 5;
```

Package

- Packages are containers for classes that are used to keep the class name space compartmentalized.
- For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.
- The package is both a **naming and a visibility control mechanism.**
- **You can define classes inside a package that are not accessible by code outside that package.**
- **You can also define class members that are only exposed to other members of the same package.**
- **This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.**

Defining a package

package pkg_name;

- Here, package is a keyword and pkg_name is the name of the package.
- For example, the following statement creates a package called MyPackage.

  package MyPackage;
- In Java a hierarchy of packages can be created. To do so, simply separate each package name from the one above it by use of a period.
- The general form of a multileveled package statement is shown here: package pkg1[.pkg2[.pkg3]];
- A package hierarchy must be reflected in the file system of Java development system.
- For example, a package declared as

package java.awt.image;

needs to be stored in java\awt\image on the Windows file system.

Ref:Program: PackageDemo

Package: Example

```
package Demo;
class PackageDemo{
     void display()
    {
        System.out.println("hello i am inside package demo");
    }
    public static void main(String args[])
    {
        PackageDemo dm=new PackageDemo();
        dm.display();
    }
}
```

Access Modifiers

- The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class.
- We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

```
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
   }
}
```

Ref: Program PrivateTest.java

Role of constructor

- If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

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

2. Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package.

- If you don't use any modifier, it is treated as **default** 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.

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

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

3. Protected: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

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

//save by B.java
package mypackB;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

4. Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

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

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

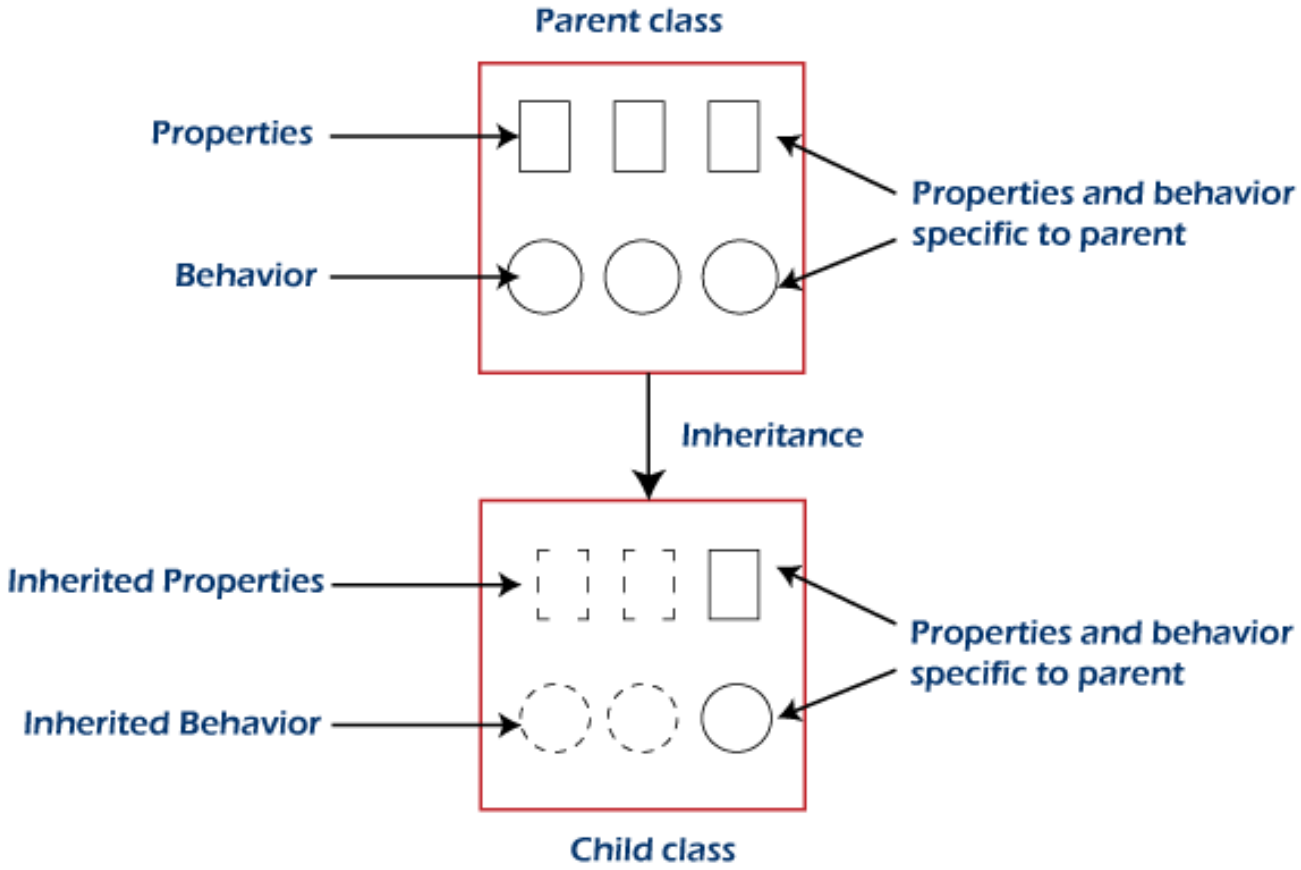
Access Modifiers

- The three access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories.
- Table below sums up the interactions

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Inheritance

- Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications.
- Using inheritance, you can create a general class that defines traits common to a set of related items.
- This class can then be inherited by other, more specific classes, each adding those things that are unique to it.
- In the terminology of Java, a class that is inherited is called a superclass.
- The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a superclass.
- It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.
- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.


Parent class
Properties
Behavior
Properties and behavior specific to parent
Inheritance
Inherited Properties
Inherited Behavior
Properties and behavior specific to parent
Child class

```
class A
{
        int i, j;
        void showij( )
        {
             System.out.println("i and j: " + i + " " + j);
         }
}
// Create a subclass by extending class A.
class B extends A
{
       int k;
      void showk( )
      {
              System.out.println("k: " + k);
      }
       void sum( )
       {
              System.out.println("i+j+k: " + (i+j+k));
      }
}
```

```
class SimpleInheritance
{
     public static void main(String args[ ])
    {
            A superOb = new A( );
            B subOb = new B(); // The superclass may be used by itself.
            superOb.i = 10;
            superOb.j = 20;
            System.out.println("Contents of superOb: ");
            superOb.showij();
             System.out.println();
    // The subclass has access to all public members of its superclass.
            subOb.i = 7;
            subOb.j = 8;
            subOb.k = 9;
            System.out.println("Contents of subOb: ");
            subOb.showij( );
             subOb.showk( );
            System.out.println( );
            System.out.println("Sum of i, j and k in subOb:");
            subOb.sum( );
    }
}
```

Important notes:

- Constructor cannot be inherited in Java.
- Private members do not get inherited in Java.
- Cyclic inheritance is not permitted in Java.
- Assign parent reference to child objects.
- Constructors get executed because of super() present in the constructor.

Types of Inheritance in java

- Java supports four types of Inheretance
- Single Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Multiple inheritance is not supported by java

1. Single Inheritance

- In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes it is also known as **simple inheritance**.

Example:

```
class parent
{
   int a=100;
   void display()
   {
      System.out.println("inside parent");
   }
}
public class SingleInheritance extends parent {
   public static void main(String args[])
   {
     SingleInheritance s=new SingleInheritance();
     System.out.println("value of a in parent="+s.a);
     s.display();

        }
}
```

2. Multi-level Inheritance

- In **multi-level inheritance**, a class is derived from a class which is also derived from another class is called multi-level inheritance. In simple words, we can say that a class that has more than one parent class is called multi-level inheritance.
- Note that the classes must be at different levels.
- Hence, there exists a single base class and single derived class but multiple intermediate base classes.

Example:

```
class getValues
{
  double principle;
  double rate;
  double time;
  void getData(double p, double r, double t)
  {
    principle=p;
    rate=r;
    time=t;
  }
  void display()
  {
    System.out.println("principle="+principle);
    System.out.println("Rate="+rate);
    System.out.println("Time="+time);
  }
}
class calculateSI extends getValues
{
    double SI()
    {
        return principle*rate*time;
    }

       }
```

```
public class multiLevelInheritance extends calculateSI
    {
    public static void main(String[] args) {
        multiLevelInheritance m=new
    multiLevelInheritance();
        m.getData(2345.6, 9.5, 3);
        m.display();
        System.out.println("Simple Intrest="+m.SI());


    }
}
```

3. Hierarchical Inheritance

- If a number of classes are derived from a single base class, it is called **hierarchical inheritance**.
- **Example:**

```
class A
{
   int a=1;
   void displayA()
   {
    System.out.println("inside A");
   }
}
class B extends A
{
    int a=2;
    void displayB()
    {
     System.out.println("inside B");
    }
}
```

```
class C extends A
{
   int a=3;
   void displayC()
   {
    System.out.println("inside C");
   }
}
class D extends A
{
   int a=4;
   void displayD()
   {
    System.out.println("inside D");
   }
}
public class hierarchicalInheritance {
   public static void main(String[] args) {
      D d=new D();
      C c=new C();
      B b=new B();
      d.displayD();
      d.displayA();
      c.displayC();
      c.displayA();
      b.displayB();
      b.displayA();

   }
}
```

4. Hybrid Inheritance

- Hybrid means consist of more than one. Hybrid inheritance is the combination of two or more types of inheritance.

Example:

```
import javax.sound.sampled.SourceDataLine;

    class GrandFather
{
   void showGrand()
   {
      System.out.println("I am grand Father");
   }
}
class Father extends GrandFather
{
   void showFather()
   {
      System.out.println("I am  Father");
   }
}
```

```
class son extends Father

{
    void show()
    {
        System.out.println("I am grand son");
    }
}
class daughter extends Father
{
    void show()
    {
        System.out.println("I am daughter");
    }
}
public class hybridInheritance {
    public static void main(String[] args) {
        daughter d=new daughter();
        d.show();
        d.showFather();
        d.showGrand();
    }
}
```

Super Keyword

super has two general forms.

- The first calls the superclass' constructor.
- The second is used to access a member of the superclass that has been hidden by a member of a subclass

→A subclass can call a constructor method defined by its superclass by use of the following form of super:

super(parameter-list);

Here, parameter-list specifies any parameters needed by the constructor in the superclass. super( ) must always be the first statement executed inside a subclass' constructor.

```
class Box{
   private double width;
   private double heigth;
   private double depth;
   //Connstructor for an oject
   Box(Box ob)
   {
      width=ob.width;
      heigth=ob.heigth;
      depth=ob.depth;

            }
   //constructor for all dimension
   Box(double w, double h, double  d)
   {
      width=w;
      heigth=h;
      depth=d;
   }
   //constructor for no dimension
   Box()
   {
      width=-1;
      heigth=-1;
      depth=-1;
   }
   //constructor for cube
   Box(double len)
   {
      width=heigth=depth=len;
   }
   //compute volume
   double volume()
   {
       return heigth*width*depth;
   }

         }
```

```
class BoxWeight extends Box{
   double weight;
   //constructor for object
   BoxWeight(BoxWeight ob)
   {
      super(ob);
      weight=ob.weight;
   }
   //Constructor for all parameters
   BoxWeight(double w, double h, double d, double m)
   {
      super(w,h,d);
      weight=m;
   }
   //constructor for no dimension
   BoxWeight()
   {
      super();
      weight=-1;
   }
   //constructor for cube
   BoxWeight(double len, double m)
   {
      super(len);
      weight=m;
   }
}
```

```
public class DemoSuper {
    public static void main(String args[])
    {
        BoxWeight mybox1=new BoxWeight(5,12);
        System.out.println(mybox1.volume());
        System.out.println(mybox1.weight);
    }

}
```

A Second Use for super

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

➢ **Progrm: UseSuper**

```
class A {
    int i;
    int j;
}
// Create a subclass by extending class A.
    class B extends A {
        int i; // this i hides the i in A
        B(int a, int b,int c) {
            super.i = a; // i in A
            i = b; // i in B
            j=c;
        }
        void show() {
            System.out.println("i in superclass: " + super.i);
            System.out.println("i in subclass: " + i);
        }
}
class UseSuper {
        public static void main(String args[]) {
        B subOb = new B(1, 2,3);
        subOb.show();
    }
}
```

Interface

- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*.
- There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
- Java Interface also **represents the IS-A relationship**.
- It cannot be instantiated just like the abstract class.
- Since Java 8, we can have **default and static methods** in an interface.
- Since Java 9, we can have **private methods** in an interface.

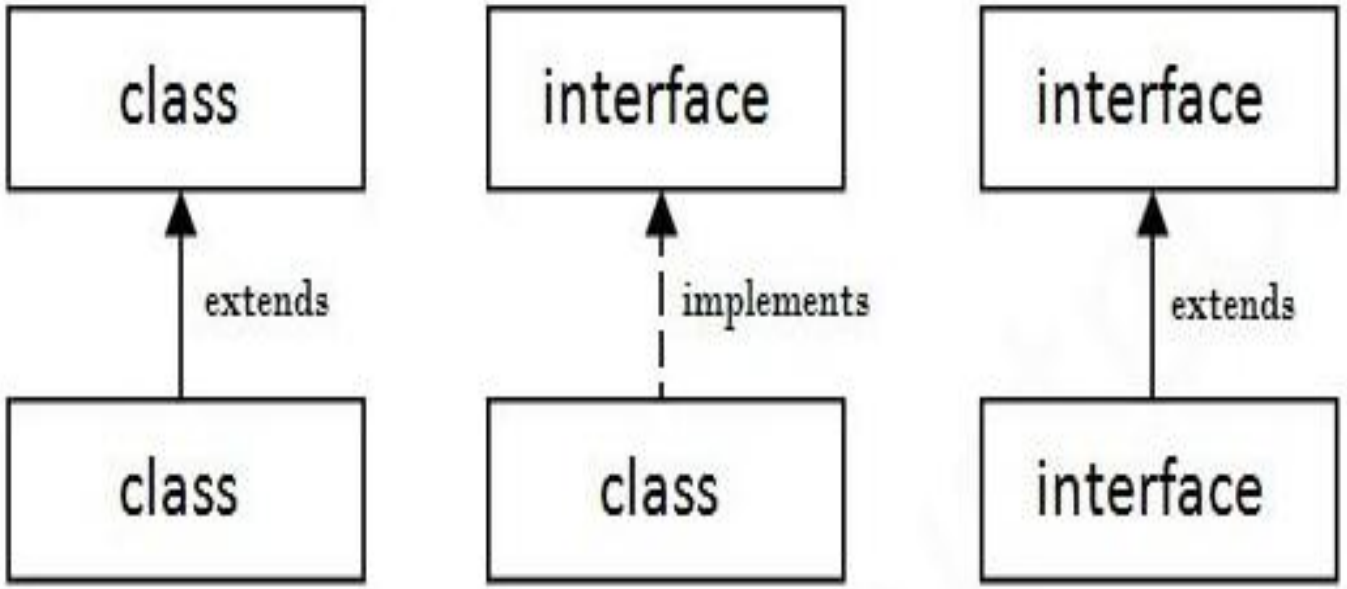
There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Declaration:

- An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
- A class that implements an interface must implement all the methods declared in the interface.

Relationship between class and interface



Example:

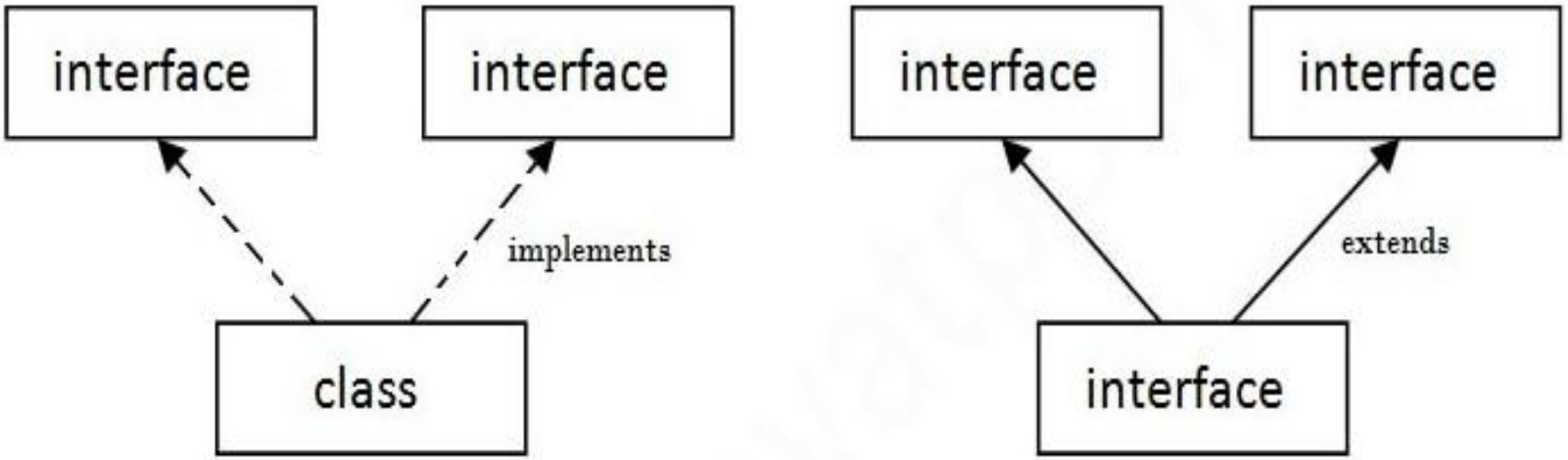
```
interface draw{
    void drawShape();
    //void abc();
}
class rectangle implements draw{
    public void drawShape()
    {
        System.out.println("rectangle is drawing");
    }
}
class triangle implements draw{
    public void drawShape()
    {
        System.out.println("triangle is drawing");
    }
}
public class interface1 {
    public static void main(String[] args) {
        triangle t=new triangle();
        rectangle r=new rectangle();
        t.drawShape();
        r.drawShape();

          }

}
```

Multiple inheritance in java

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



Example:

```
interface Printable{
    void print();
    }
interface Showable{
    void show();
}
public class multipleInheritance implements Printable, Showable {
    public void print()
    {
       System.out.println("printing");
    }
    public void show()
    {
       System.out.println("showing");
    }
    public static void main(String[] args) {
        multipleInheritance m=new multipleInheritance();
        m.print();
        m.show();
    }

}
```

Interface inheritance

A class implements an interface, but one interface extends another interface

Example:

```
import javax.xml.transform.Source;

      interface printable{
   void print();
}
interface showable extends printable{
   void show();
}
public class InterfaceInheritance implements showable {
   public void show()
   {
      System.out.println("showing");
   }
   public void print()
   {
     System.out.println("printing");
   }
   public static void main(String[] args) {
       InterfaceInheritance i=new InterfaceInheritance();
       i.show();
       i.print();
   }
}
```

Nested Interface in Java

- Note: An interface can have another interface which is known as a nested interface. We will learn it in detail in the nested classes chapter. For example:

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

Java Inner Class

- **Java inner class** or nested class is a class that is declared inside the class or interface.
- We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.
- Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax

```
class Java_Outer_class
{
 //code
 class Java_Inner_class
     {
       //code
      }
}
```

Advantages

- Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class,** including private.
- Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
- **Code Optimization**: It requires less code to write.

Need of Java Inner class

- Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.
- If all the class objects are a part of the outer object then it is easier to nest that class inside the outer class. That way all the outer class can access all the objects of the inner class.

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing an interface or extending class. The java compiler decides its name.
Local Inner Class	A class was created within the method.
Static Nested Class	A static class was created within the class.
Nested Interface	An interface created within class or interface.

Homework.

Explain following with suitable examples.

- Nested Inner Class
- Method Local Inner Classes
- Static Nested Classes
- Anonymous Inner Classes