

PROGRAMMING IN JAVA

INTERMEDIATE LEVEL

Programming in Java

The difference between an interpreter and a compiler is given below:

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers



Figure: Compiler



Figure: Interpreter

C++ vs Java

There are many differences and similarities between the **C++** programming language and **Java**. A list of top differences between C++ and Java are given below:

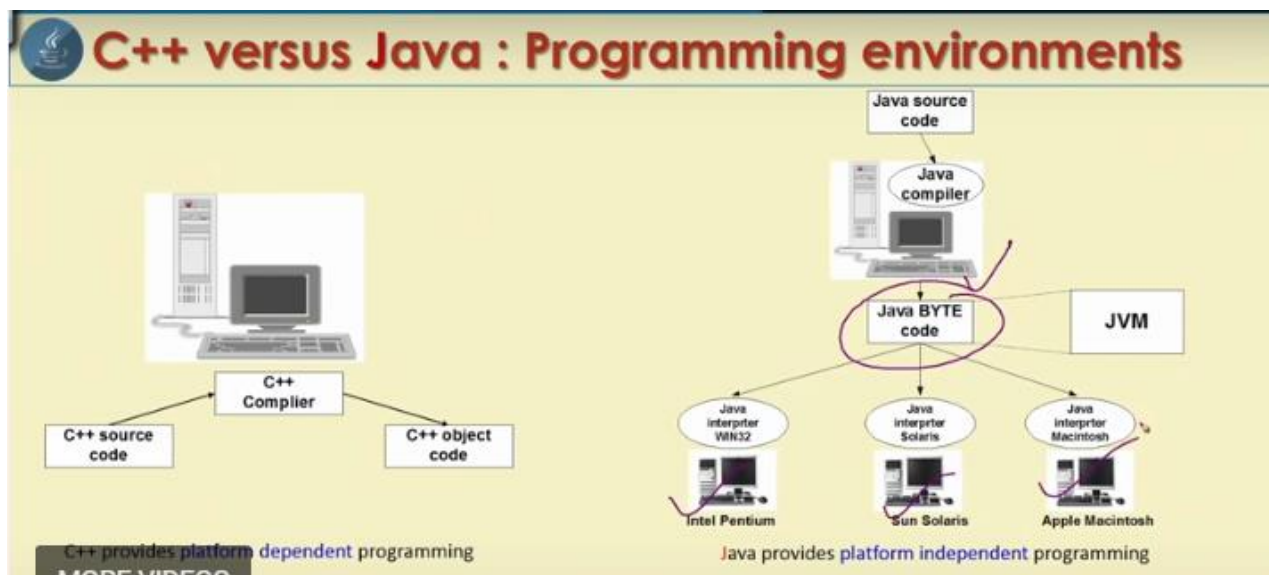
Comparison Index	C++	Java
Platform-independent	C++ is platform dependent.	Java is platform independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed with a goal of being easy to use and accessible to a broader audience.
Goto	C++ supports the goto statement.	Java doesn't support the goto statement.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by interfaces in java.
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.
Compiler and Interpreter	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into	Java uses compiler and interpreter both. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at

	machine code so, C++ is platform dependent.	runtime and produces output. Java is interpreted that is why it is platform independent.
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Documentation comment	C++ doesn't support documentation comment.	Java supports documentation comment (<code>/** ... */</code>) to create documentation for java source code.
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
unsigned right shift >>>	C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator.
Inheritance Tree	C++ creates a new inheritance tree always.	Java uses a single inheritance tree always because all classes are the child of Object class in java. The object class is the root of the inheritance tree in java.
Hardware	C++ is nearer to hardware.	Java is not so interactive with hardware.

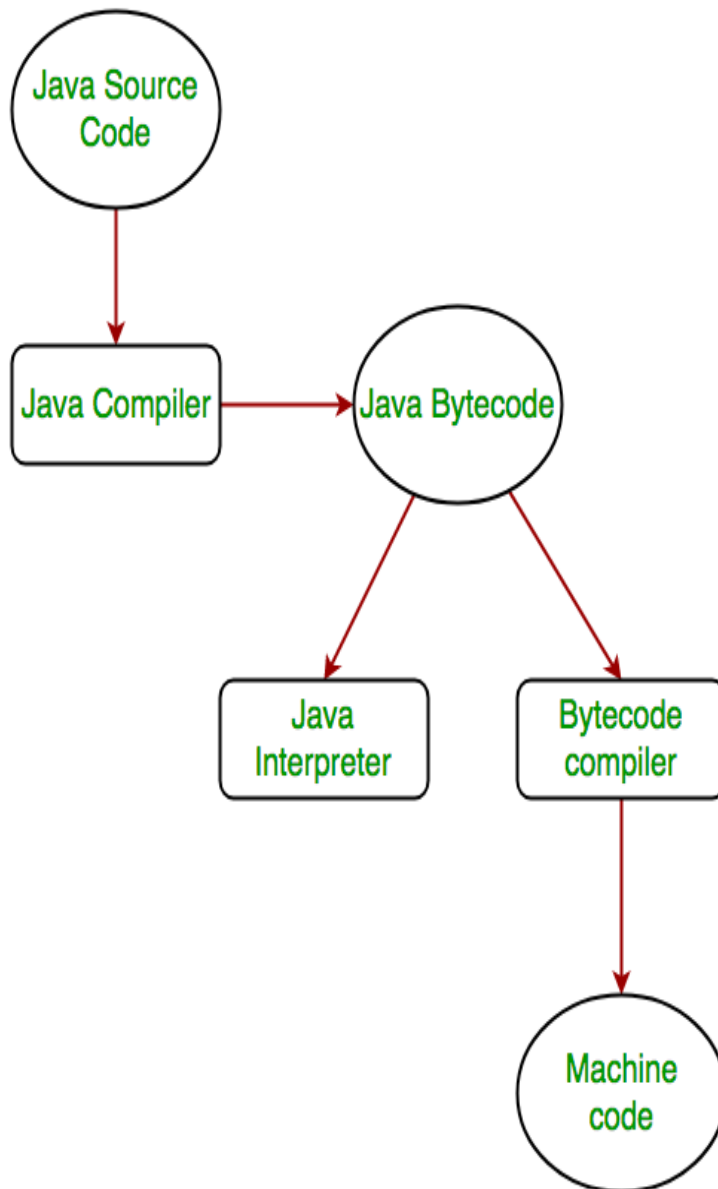
Object-oriented	C++ is an object-oriented language. However, in C language, single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object.
------------------------	--	--

Note

- Java doesn't support default arguments like C++.
- Java does not support header files like C++. Java uses the import keyword to include different classes and methods



Why java is a platform independent language?



Step by step Execution of Java Program:

1. Whenever, a program is written in JAVA, the javac compiles it.
2. The result of the JAVA compiler is the **.class file or the bytecode** and not the machine native code (unlike C compiler).
3. The bytecode generated is a non-executable code and needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM.
4. And finally program runs to give the desired output.

Execution of c++ program

When you write program in C/C++ and when you compile it, it is directly converted into machine readable language(.exe). This .exe file generated is specific to the operating system i.e, when you compile program in windows OS, the .exe file generated for that program is specific to only windows OS and cannot be made to run in UNIX OS.

That's why C/C++ programs are platform dependent.

JDK JRE AND JVM

JVM (JAVA VIRTUAL MACHINE)

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JRE (JAVA RUNTIME ENVIRONMENT)

The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

JDK (JAVA DEVELOPMENT KIT)

The JDK is a collection of the JVM, JRE and libraries. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

NOTE

JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent.

Refer to: <https://www.javatpoint.com/internal-details-of-jvm>

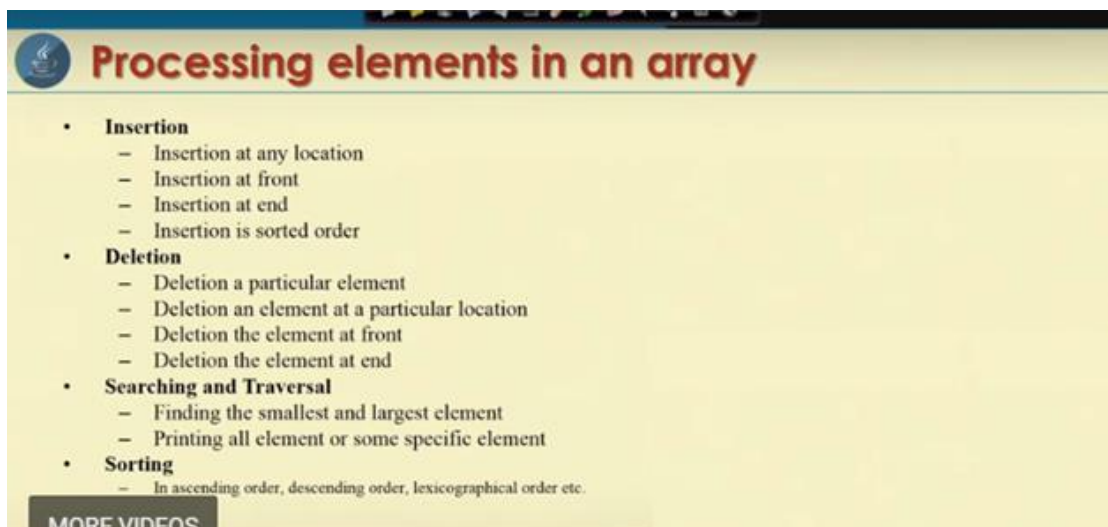
Java is platform independent but JVM is platform dependent

In Java, the main point here is that the JVM depends on the operating system – so if you are running Mac OS X you will have a different JVM than if you are running Windows or some other operating system. This fact can be verified by trying to download the JVM for your particular machine – when trying to download it, you will given a list of JVM's corresponding to different operating systems, and you will obviously pick whichever JVM is targeted for the operating system that you are running. So we can conclude that JVM is platform dependent and it is the reason why Java is able to become “Platform Independent”.

Important Points:

- In the case of Java, **it is the magic of Bytecode that makes it platform independent.**
- This adds to an important feature in the JAVA language termed as **portability**. Every system has its own JVM which gets installed automatically when the jdk software is installed. For every operating system separate JVM is available which is capable to read the .class file or byte code.
- An important point to be noted is that while **JAVA is platform-independent language, the JVM is platform-dependent.** Different JVM is designed for different OS and byte code is able to run on different OS.

[Array questions to be done](#)

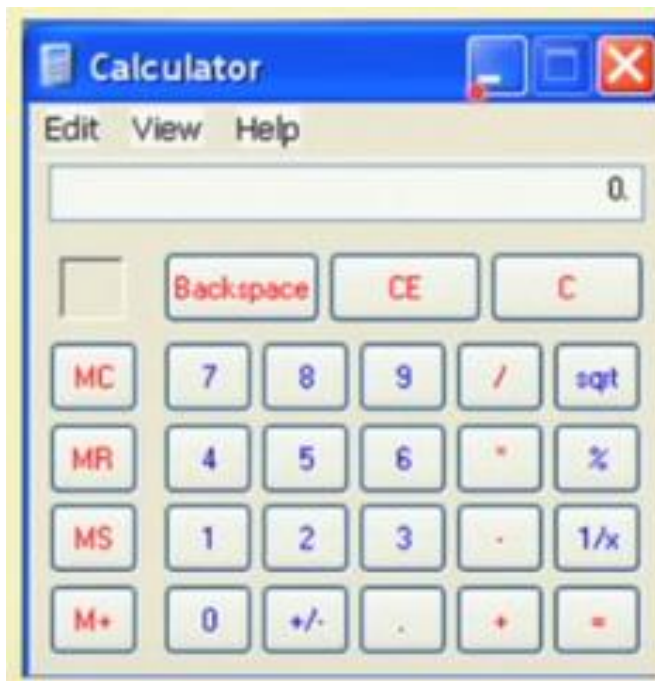


Note the types of Array

- 1-D array
- 2-D array
- Variable sized 2-D array

The two types of java programming are

1. Java applications
2. Java applets
 1. Java applications-It is similar to other kinds of program in C, C++ etc. to solve a problem.
 2. Java applet-It is a program that appears embedded in a web document and applets come into effect when the browser browses the web page. It is a small application made in java. Example calculator.



Java Programming

CLASS

A **class** is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers:** A class can be public or has default access.
2. **Class name:** The name should begin with an initial letter.
3. **Superclass (if any):** The name of the class's parent (superclass), if any, preceded by the keyword `extends`. A class can only extend (subclass) one parent.
4. **Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword `implements`. A class can implement more than one interface.
5. **Body:** The class body surrounded by braces, `{ }`.

Constructors are used for initializing new objects. Implicit return type of the constructor is the class itself. It can also be overloaded.

OBJECT

It is a basic unit of Object-Oriented Programming and represents the real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of:

1. **State:** It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior:** It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Example of an object: dog

ENCAPSULATION

The wrapping of data into a singular unit. It is the mechanism that binds the data together and help in showing only useful things so also known as data hiding.

As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.

<pre> import java.util.*; class Lcm { int x; int y; public Lcm(int a,int b) { x=a; y=b; } public int calculate() { int i=x>y?x:y; while(true) { if(i%x==0 && i%y==0) { break; } i++; } return i; } } </pre>	<pre> import java.util.*; class Hcf { int x; int y; public Hcf(int a,int b) { x=a; y=b; } public int calculate() { int hcf = 1; for(int i = 1; i <= x && i <= y; ++i) { if(x % i==0 && y % i==0) { hcf = i; } } return hcf; } } </pre>
<pre> import java.util.*; class Lcm_hcf { public static void main() { Scanner in=new Scanner(System.in); System.out.println("Enter number 1"); int x=in.nextInt(); System.out.println("Enter number 2"); int y=in.nextInt(); Lcm ob1=new Lcm(x,y); Hcf ob2=new Hcf(x,y); System.out.println("Lcm of two numbers is "+ob1.calculate()); System.out.println("Hcf of two numbers is "+ob2.calculate()); } } </pre>	

Two different objects of two different class is used in a same class. A program should have a main function or else it will show a runtime error. The above three programs can be in a same file or different file.

USE OF THIS KEYWORD

```
import java.util.*;

class This_keyword
{
    int x;
    int y;
    int z;
    public This_keyword(int z)
    {
        this.z=z;
    }
    public This_keyword(int x,int y)
    {
        this(25);
        this.x=x;
        this.y=y;
    }
    public void display(int x,int y)
    {
        System.out.println("Without this keyword \n x="+x+"\ty="+y);
        this.this_display();
    }
    public void this_display()
    {
        System.out.println("With this keyword \n x="+this.x+"\ty="+this.y);
    }
    public static void main(String []args)
    {
        This_keyword ob=new This_keyword(13,17);
        ob.display(5,24);
        System.out.println("Value of z="+ob.z);
    }
}
```

THIS keyword is a reference variable of that instance. It is used to give values to member variables of class. It can also be used to call different functions of same class. It can also be used to call other constructors from a constructor. While calling the other constructor from this keyword, it should be the first statement of the constructor. (Nesting of constructor calling)

[REFER TO SUPER KEYWORD](#)

In this statement,

`"Public static void main(String args[])"`

Static is used to allow main() to be called without having to instantiate the instance of class.

Main is used as the name of the method of class which is searched by the JVM as the starting point of the class.

Args[] is an array to store objects of class String. Any other word or letter may also be used. Java sees everything as String objects. It helps to read an input and then store into the array args as String objects.

System.out.printf("number is %d",n); is used to display formatted integers on the contrary to System.out.println(); or System.out.print();

Object declaration can also be done in other methods. It can even be done globally outside the main function. ***The main can only access a static object.***

ENHANCED FOR LOOP

<pre>for (int i = 0; i < Array.length; i++) { System.out.println(Array[i]); }</pre>	<pre>for (int i : Array) { System.out.println(i); }</pre>
--	---

In enhanced for loop writing System.out.println(Array[i]); inside the loop will give an error.

OTHER INPUT METHODS

1. Using DATA INPUT STREAM

```
import java.io.*;

class Data_Input_Stream
{
    public static void main()throws IOException
    {
        DataInputStream in=new DataInputStream(System.in);

        System.out.println("Enter a number");
        int n;
        n=Integer.parseInt(in.readLine());

        System.out.printf("You entered:%d",n);
    }
}
```

2. Using BUFFERED READER

```
import java.io.*;

class Buffered_Reader
{
    public static void main()throws IOException
    {
        BufferedReader inp = new BufferedReader (new
        InputStreamReader(System.in));
        System.out.println("Enter a number");
        int n;
        n=Integer.parseInt(inp.readLine());

        System.out.printf("You entered:%d",n);
    }
}
```

In the above programs it is necessary to write "throws IOException".

3. Using SCANNER OBJECT

```
import java.util.*;

class Scanner_Input
{
    public static void main()
    {
        Scanner in=new Scanner(System.in);

        System.out.println("Enter a number");
        int n=in.nextInt();
        System.out.println("You entered:"+n);
    }
}
```

However String args[] written in main function can also be used.

ARRAY QUESTIONS

1. INSERTION

```
import java.util.*;

class Array
{
    public static void main(String args[])
    {
        Scanner in=new Scanner(System.in);

        System.out.println("Enter the size of array ");
        int size=in.nextInt();

        int a[]=new int[size];
        int c=0;int p=0;int pos=0;int n=0;int nn=0;
        for(int i=0;i<a.length;i++)
        {
            n=in.nextInt();
            if(n!=0)
            {
                a[i]=n;
                c++;
            }
        }

        if(c==5)
            System.out.println("Element cannot be Entered");
        else
        {
            System.out.println("Enter a number to be intered");
            n=in.nextInt();
            System.out.println("Enter your choice");
            System.out.println("1=Front");
            System.out.println("2=Enter Position");
            System.out.println("3=End");
            System.out.println("4=Sorted Insertion");
            nn=in.nextInt();

            for(int i=0;i<a.length;i++)
            {
                if(a[i]==0)
                {
                    pos=i;
                    break;
                }
            }
        }
    }
}
```

```

    }
}
switch(nn)
{
    case 1 :p=0;break;
    case 2 :
    {
        p=in.nextInt();
        p--;
        break;
    }
    case 3 :p=a.length-1;break;
    case 4 :p=pos;break;
    default:System.out.println("Wrong input");
}
}

if(pos<p)
{
    for(int i=pos;i<a.length-1;i++)
        a[i]=a[i+1];

    a[p]=n;
}
else if(pos>p)
{
    for(int i=pos;i>0;i--)
        a[i]=a[i-1];

    a[p]=n;
}
else
    a[p]=n;

if(nn==4)
{
    int l = a.length;
    for (int i = 0; i < l-1; i++)
    {
        for (int j = 0; j < l-i-1; j++)
        {
            if (a[j] > a[j+1])
            {
                int temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}
}

```

```

    }
    }
    }
    System.out.println("Your final array is");
    for(int i=0;i<a.length;i++)
        System.out.println(a[i]);
    }
}

```

2. DELETION

```

import java.util.*;

class Arrays
{
    public static void main(String args[])
    {
        Scanner in=new Scanner(System.in);

        System.out.println("Enter the size of array ");
        int size=in.nextInt();

        int a[]=new int[size];
        int c=0;int p=0;int pos=0;int n=0;int nn=0;
        for(int i=0;i<a.length;i++)
            a[i]=in.nextInt();

        System.out.println("Enter your choice");
        System.out.println("1=Front");
        System.out.println("2=Enter Position");
        System.out.println("3=End");
        System.out.println("4=Particular Element Deletion");
        nn=in.nextInt();

        switch(nn)
        {
            case 1 :pos=0;break;
            case 2 :
            {
                pos=in.nextInt();
                pos--;
                break;
            }
            case 3 :pos=a.length-1;break;

```

```

case 4 :
{
    System.out.println("Enter element to be deleted");
    int del=in.nextInt();
    boolean check=false;

    for(int i=0;i<a.length;i++)
    {
        if(a[i]==del)
        {
            pos=i;
            check=true;
            break;
        }
    }
    if(check==false)
    {
        System.out.println("No Such Element");
        System.out.println("Your final array is");
        for(int i=0;i<a.length;i++)
            System.out.println(a[i]);
        System.exit(0);
    }
    break;
}
default: System.out.println("Wrong input");
}

for(int i=pos;i<a.length-1;i++)
    a[i]=a[i+1];

a[a.length-1]=0;
System.out.println("Your final array is");
for(int i=0;i<a.length;i++)
    System.out.println(a[i]);
}
}

```

3. Search

a. Linear Search

```
import java.util.*;

class Linear_Search
{
    int size;
    int a[];
    int search;
    Scanner in=new Scanner(System.in);
    public Linear_Search(int n)
    {
        size=n;
        a=new int[size];
    }
    public void input()
    {
        System.out.println("Enter the elements of the array:");
        for(int i=0;i<size;i++)
            a[i]=in.nextInt();

        System.out.println("Enter the search element:");
        search=in.nextInt();
    }
    public int search()
    {
        for(int i=0;i<size;i++)
        {
            if(a[i]==search)
                return i;
        }
        return -1;
    }
    public static void main()
    {
        Scanner in=new Scanner(System.in);

        System.out.println("Enter the size of the array:");
        int n=in.nextInt();

        Linear_Search ob=new Linear_Search(n);
        ob.input();
        int index=ob.search();
        System.out.println(index!=-1?"Search element found at "+index+" index":"Search element not found");
    }
}
```



```

    }
}

```

These search algorithms require array to be sorted in a particular order.

b. Binary Search

```

import java.util.*;

class Binary_Search
{
    int size;
    int a[];
    int search;
    Scanner in=new Scanner(System.in);
    public Binary_Search(int n)
    {
        size=n;
        a=new int[size];
    }
    public void input()
    {
        System.out.println("Enter the elements of the array in
ascending order or descending order:");
        for(int i=0;i<size;i++)
            a[i]=in.nextInt();

        System.out.println("Enter the search element:");
        search=in.nextInt();
    }
    public int search()
    {
        int f=1;
        int z=1;
        for(int i=0,j=a.length-1,k=0;k<=a.length;k++)
        {
            z=(int)(j+i)/2;
            if(a[z]==search)
            {
                f=0;
                return (z+1);
            }
            else if(a[z]<search)
                i=z+1;
            else

```

```

        j=z-1;
    }
    return -1;
}
public static void main()
{
    Scanner in=new Scanner(System.in);

    System.out.println("Enter the size of the array:");
    int n=in.nextInt();

    Binary_Search ob=new Binary_Search(n);
    ob.input();
    int index=ob.search();
    System.out.println(index!=-1?"Search element found at
    "+index+" index":"Search element not found");

}
}

```

c. Jump Search

```

import java.util.*;

class Jump_Search
{
    int size;
    int a[];
    int search;
    Scanner in=new Scanner(System.in);
    public Jump_Search(int n)
    {
        size=n;
        a=new int[size];
    }
    public void input()
    {
        System.out.println("Enter the elements of the array in
        ascending order or descending order:");
        for(int i=0;i<size;i++)
            a[i]=in.nextInt();

        System.out.println("Enter the search element:");
        search=in.nextInt();
    }
    public int search()

```

```

{
    int jump=(int)Math.floor(Math.sqrt(size));
    int prev=0;
    while(a[Math.min(jump,size)-1]<search)
    {
        prev=jump;
        jump+=(int)Math.floor(Math.sqrt(size));

        if(prev>=size)
            return -1;
    }

    while(a[prev]<search)
    {
        prev++;
        if(prev==Math.min(jump,size))
            return -1;
    }

    if(a[prev]==search)
        return prev;

    return -1;
}

public static void main()
{
    Scanner in=new Scanner(System.in);

    System.out.println("Enter the size of the array:");
    int n=in.nextInt();

    Jump_Search ob=new Jump_Search(n);
    ob.input();
    int index=ob.search();
    System.out.println(index!=-1?"Search element found at
    "+index+" index":"Search element not found");
}
}

```

d. Interpolation Search

```

import java.util.*;

class Interpolation_Search
{
    int size;
    int a[];
    int search;
    Scanner in=new Scanner(System.in);
    public Interpolation_Search(int n)
    {
        size=n;
        a=new int[size];
    }
    public void input()
    {
        System.out.println("Enter the elements of the array in
ascending order or descending order:");
        for(int i=0;i<size;i++)
            a[i]=in.nextInt();

        System.out.println("Enter the search element:");
        search=in.nextInt();
    }
    public int search()
    {
        int l=0;
        int h=size-1;
        while(l<=h&& a[l]<=search&& a[h]>=search)
        {
            if(h==l)
            {
                if(a[l]==search) return l;
                return -1;
            }

            int pos=l+(int)((((h-l)/(a[h]-a[l]))*(search-a[l])));

            if(a[pos]==search) return pos;
            if(a[pos]<search) l=pos+1;
            if(a[pos]>search) h=pos-1;
        }
        return -1;
    }
}

```

```
public static void main()
{
    Scanner in=new Scanner(System.in);

    System.out.println("Enter the size of the array:");
    int n=in.nextInt();

    Interpolation_Search ob=new Interpolation_Search(n);
    ob.input();
    int index=ob.search();
    System.out.println(index!=-1?"Search element found at
    "+index+" index":"Search element not found");

}
}
```

4. Sorting

a. Selection Sort

```
import java.util.*;

class Selection_Sort
{
    int size;
    int a[];

    Scanner in=new Scanner(System.in);
    public Selection_Sort(int n)
    {
        size=n;
        a=new int[size];
    }
    public void input()
    {
        System.out.println("Enter the elements of the array:");
        for(int i=0;i<size;i++)
            a[i]=in.nextInt();
    }
    public void sort()
    {
        for(int i=0;i<size-1;i++)
        {
            int s=i;
            for(int j=i+1;j<size;j++)
                if(a[j]<a[s])
                    s=j;

            int temp=a[s];
            a[s]=a[i];
            a[i]=temp;
        }
    }
    public void display()
    {
        System.out.println("Sorted Array:");
        for(int i:a)
            System.out.println(i+"\t");
    }
    public static void main()
    {
        Scanner in=new Scanner(System.in);
```



```

        System.out.println("Enter the size of the array:");
        int n=in.nextInt();

        Selection_Sort ob=new Selection_Sort(n);
        ob.input();
        ob.sort();
        ob.display();

    }
}

```

b. Bubble Sort

```

import java.util.*;

class Bubble_Sort
{
    int size;
    int a[];

    Scanner in=new Scanner(System.in);
    public Bubble_Sort(int n)
    {
        size=n;
        a=new int[size];
    }
    public void input()
    {
        System.out.println("Enter the elements of the array:");
        for(int i=0;i<size;i++)
            a[i]=in.nextInt();
    }
    public void sort()
    {
        for(int i=0;i<size;i++)
        {
            for(int j=0;j<size-1-i;j++)
            {
                if(a[j]>a[j+1])
                {
                    int t=a[j];
                    a[j]=a[j+1];
                    a[j+1]=t;
                }
            }
        }
    }
}

```

```

    }
    public void display()
    {
        System.out.println("Sorted Array:");
        for(int i:a)
            System.out.println(i+"\t");
    }
    public static void main()
    {
        Scanner in=new Scanner(System.in);

        System.out.println("Enter the size of the array:");
        int n=in.nextInt();

        Bubble_Sort ob=new Bubble_Sort(n);
        ob.input();
        ob.sort();
        ob.display();

    }
}

```

c. Insertion Sort

```

import java.util.*;

class Insertion_sort
{
    int size;
    int a[];

    Scanner in=new Scanner(System.in);
    public Insertion_sort(int n)
    {
        size=n;
        a=new int[size];
    }
    public void input()
    {
        System.out.println("Enter the elements of the array:");
        for(int i=0;i<size;i++)
            a[i]=in.nextInt();
    }
    public void sort()
    {
        for(int i=1;i<size;i++)

```

```

    {
        int key=a[i];
        int j=i-1;

        while(j>=0&& a[j]>key)
        {
            a[j+1]=a[j];
            j=j-1;
        }
        a[j+1]=key;
    }
}
public void display()
{
    System.out.println("Sorted Array:");
    for(int i:a)
        System.out.println(i+"\t");
}
public static void main()
{
    Scanner in=new Scanner(System.in);

    System.out.println("Enter the size of the array:");
    int n=in.nextInt();

    Insertion_sort ob=new Insertion_sort(n);
    ob.input();
    ob.sort();
    ob.display();
}
}

```

d. Cocktail Sort

```

import java.util.*;

class Cocktail_Sort
{
    int size;
    int a[];

    Scanner in=new Scanner(System.in);
    public Cocktail_Sort(int n)
    {
        size=n;
    }
}

```

```
a=new int[size];
}
public void input()
{
    System.out.println("Enter the elements of the array:");
    for(int i=0;i<size;i++)
        a[i]=in.nextInt();
}
public void sort()
{
    boolean swap=true;
    int s=0;
    int e=a.length;

    while(swap=true)
    {
        swap=false;

        for(int i=s;i<e-1;i++)
        {
            if(a[i]>a[i+1])
            {
                int temp=a[i];
                a[i]=a[i+1];
                a[i+1]=temp;
                swap=true;
            }
        }

        if(swap==false)
            break;

        swap=false;
        e=e-1;
        for(int i=e-1;i>=s;i--)
        {
            if(a[i]>a[i+1])
            {
                int temp=a[i];
                a[i]=a[i+1];
                a[i+1]=temp;
                swap=true;
            }
        }
        s=s+1;
    }
}
```

```

    }
    public void display()
    {
        System.out.println("Sorted Array:");
        for(int i:a)
            System.out.println(i+"\t");
    }
    public static void main()
    {
        Scanner in=new Scanner(System.in);

        System.out.println("Enter the size of the array:");
        int n=in.nextInt();

        Cocktail_Sort ob=new Cocktail_Sort(n);
        ob.input();
        ob.sort();
        ob.display();

    }
}

```

e. Merge Sort

```

import java.util.*;

class Merge_Sort
{
    int size;
    int a[];

    Scanner in=new Scanner(System.in);
    public Merge_Sort(int n)
    {
        size=n;
        a=new int[size];
    }
    public void input()
    {
        System.out.println("Enter the elements of the array:");
        for(int i=0;i<size;i++)
            a[i]=in.nextInt();
        sort(a,0,a.length-1);
    }
    public void sort(int a[],int l,int r)

```

```

{
    if(l<r)
    {
        int m=(l+r)/2;

        sort(a,l,m);
        sort(a,m+1,l);

        merge(a,l,m,r);
    }
}
public void merge(int a[],int l,int m,int r)
{
    int n1=m+1-l;
    int n2=r-m;

    int X[]=new int [n1];
    int Y[]=new int [n2];

    for(int i=0;i<n1;i++)
    X[i]=a[i+l];
    for(int j=0;j<n2;j++)
    Y[j]=a[m+j+1];

    int i = 0,j = 0,k = l;
    while (i < n1 && j < n2)
    {
        if (X[i] <= Y[j])
        {
            a[k] = X[i];
            i++;
        }
        else
        {
            a[k] = Y[j];
            j++;
        }
        k++;
    }

    while (i < n1)
    {
        a[k] = X[i];
        i++;
        k++;
    }
}

```



```
        while (j < n2)
        {
            a[k] = Y[j];
            j++;
            k++;
        }
    }
    public void display()
    {
        System.out.println("Sorted Array:");
        for(int i:a)
            System.out.println(i+"\t");
    }
    public static void main()
    {
        Scanner in=new Scanner(System.in);

        System.out.println("Enter the size of the array:");
        int n=in.nextInt();

        Merge_Sort ob=new Merge_Sort(n);
        ob.input();

        ob.display();

    }
}
```

INHERITANCE


The mechanism in java by which one class is allowed to inherit the features (data-members and methods) of another class is called INHERITANCE.

- **Super Class:** The class whose features are inherited is known as super class (or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as sub class (or a derived class, extended class, or child class).
- **Reusability:** The mechanism which facilitates you to re-use the data and methods of existing class is called reusability.

The keyword used for inheritance is EXTENDS.

The three types of inheritance are only possible in java programming language.

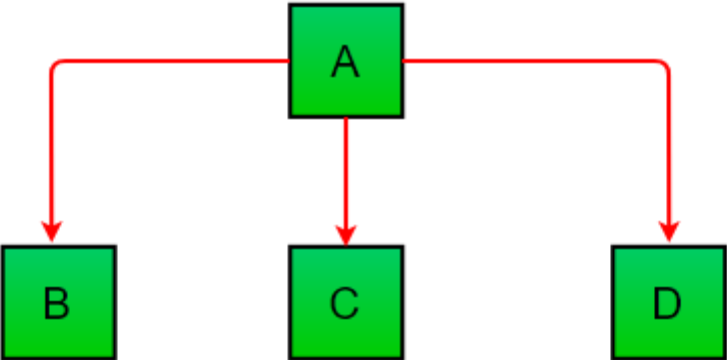
1. Single Inheritance

<pre>import java.util.*; class A { int x=13; public void display() { System.out.println("I am A"); } }</pre>	 <p>Single Inheritance</p>
<pre>class B extends A { public void output() { System.out.println("I am B"); System.out.println("x="+x); } public static void main() { B ob=new B(); ob.display(); ob.output(); } }</pre>	

2. Multilevel Inheritance

<pre>import java.util.*; class A { int x=13; public void display() { System.out.println("I am A"); } }</pre>	<pre>graph TD A[A] --> B[B] B --> C[C]</pre>
<pre>class B extends A { public void output() { System.out.println("I am B"); System.out.println("x="+x); } }</pre>	
<pre>class C extends B { public void out() { System.out.println("I am C"); System.out.println("x="+x); } public static void main() { C ob=new C(); ob.display(); ob.output(); ob.out(); } }</pre>	

3. Hierarchical Inheritance

<pre>import java.util.*; class A { int x=13; public void display() { System.out.println("I am A"); } }</pre>	<pre>class B extends A { public void output() { System.out.println("I am B"); } public static void main() { B ob=new B(); ob.display(); ob.output(); } }</pre>
<pre>class C extends A { public void out() { System.out.println("I am C"); System.out.println("x="+x); } public static void main() { C ob=new C(); ob.display(); ob.out(); } }</pre>	<pre>class D extends A { public void print() { System.out.println("I am D"); System.out.println("x="+x); } public static void main() { D ob=new D(); ob.display(); ob.print(); } }</pre>
 <pre> graph TD A[A] --> B[B] A --> C[C] A --> D[D] </pre> <p style="text-align: center; color: red;">Hierarchical Inheritance</p>	

Other types of inheritance can be achieved through interfaces.

POLYMORPHISM

The concept by which we can perform a single task using two different ways is called Polymorphism.

Method Overloading

```
import java.util.*;
class FunctionOverloading
{
    public int maths(int x, int y)
    {
        return (x + y);
    }
    public int maths(int x, int y, int z)
    {
        return (x + y + z);
    }

    public double maths(double x, double y)
    {
        return (x + y);
    }

    public static void main(String args[])
    {
        FunctionOverloading ob = new FunctionOverloading();
        System.out.println(ob.maths(10,      17)+"\t"+ob.maths(10,      17,
30)+"\t"+ob.maths(10.5, 17.5));
    }
}
```

Method Overriding (Runtime Polymorphism)

It is used to provide special implementation of a method which is already provided by a superclass.

<pre>import java.util.*; class A { int x=13; public void display() { System.out.println("I am A"); } }</pre>	<pre>import java.util.*; class E extends A { int x=100; public void display() { System.out.println("I am E"); } public static void main() { System.out.println("E ob=new E();\nob.display();"); } }</pre>
---	--

	<pre> E ob=new E(); ob.display(); System.out.println(); System.out.println("A obj=new A();\nobj.display()"); A obj=new A(); obj.display(); System.out.println(); } }</pre>
--	--

SUPER keyword

It is used to reference a parent class instance members or methods.

✚ **Super keyword used to call instance members.**

<pre> import java.util.*; class A { int x=13; public void display() { System.out.println("I am A"); } }</pre>	<pre> import java.util.*; class E extends A { int x=100; public void display() { System.out.println("I am E"); System.out.println("X="+x); System.out.println("super.X="+super.x); } public static void main() { System.out.println("E ob=new E();\nob.display()"); E ob=new E(); ob.display(); System.out.println(); System.out.println("A obj=new A();\nobj.display()"); A obj=new A(); obj.display(); System.out.println(); } }</pre>
--	---

Super keyword used to call methods

<pre>import java.util.*; class A { int x=13; public void display() { System.out.println("I am A"); } }</pre>	<pre>import java.util.*; class E extends A { int x=100; public void display() { System.out.println("I am E"); System.out.println("X="+x); } System.out.println("super.X="+super.x); super.display(); } public static void main() { System.out.println("E ob=new E();\nob.display();"); E ob=new E(); ob.display(); System.out.println(); System.out.println("A obj=new A();\nobj.display();"); A obj=new A(); obj.display(); System.out.println(); } }</pre>
---	--

✚ Used to call the constructor of parent class

<pre>import java.util.*; class A { int x; public A() { System.out.println("Constructor of class A"); x=13; } public void display() { System.out.println("I am A"); } }</pre>	<pre>import java.util.*; class E extends A { int x=100; public E() { super(); System.out.println("Constructor of class E"); } public void display() { System.out.println("I am E"); System.out.println("X="+x); System.out.println("super.X="+super.x); super.display(); } public static void main() { System.out.println("E ob=new E();\nob.display();"); E ob=new E(); ob.display(); System.out.println(); System.out.println("A obj=new A();\nobj.display();"); A obj=new A(); obj.display(); System.out.println(); } }</pre>
--	---

✚ Used to call the parameterized constructor of parent class

<pre> import java.util.*; class A { int x; public A() { System.out.println("Constructor of class A"); x=13; } public A(int b) { x=b; } public void display() { System.out.println("I am A"); } } </pre>	<pre> import java.util.*; class E extends A { int x=100; public E() { super(256); System.out.println("Constructor of class E"); } public void display() { System.out.println("I am E"); System.out.println("X="+x); } System.out.println("super.X="+super.x); super.display(); } public static void main() { System.out.println("E ob=new E();\nob.display();"); E ob=new E(); ob.display(); System.out.println(); System.out.println("A obj=new A();\nobj.display();"); A obj=new A(); obj.display(); System.out.println(); } } </pre>
--	--

[REFER TO THIS KEYWORD](#)

Data Abstraction

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

[See abstract class in java](#)

FINAL keyword in java

final keyword is used in different contexts. *Final* is a non-access modifier applicable **only to a variable, a method or a class**. Following are different contexts where final is used.

A final

- Variable - cannot be accessed in sub class
- Method - cannot be called from sub class object
- Class - cannot be sub classed.

ACCESS MODIFIERS IN JAVA

It specifies the scope of a method, constructor, variable or class in java. The four types of access modifiers in java are

- Default-Visible to anywhere in package.
- Public- Visible everywhere.
- Private-Visible to the class only and cannot be inherited.
- Protected-Visible only to the inherited-class.

As the name suggests the default access modifier in java is 'DEFAULT'.

Access Levels Modifier	CLASS	PACKAGE	SUB-CLASS	EVERYWHERE
PUBLIC	✓	✓	✓	✓
PROTECTED	✓	✓	✓	✗
DEFAULT	✓	✓	✗	✗
PRIVATE	✓	✗	✗	✗

PRIVATE ACCESS MODIFIER

```

1 class X
2 {
3     private int num=13;
4     private void output()
5     {
6         System.out.println("num="+num);
7     }
8 }
9
10 class Private_Inheritance
11 {
12     public static void main()
13     {
14         X ob=new X();
15         System.out.println(ob.num);
16         ob.output();
17     }
18 }

```

num has private access in X

output() has private access in X

PRIVATE ACCESS MODIFIER WITH PRIVATE CONSTRUCTOR

```
class X
{
    private int num=13;
    private X(){ }
    private void output()
    {
        System.out.println("num="+num);
    }
}
```

```
class Private_Inheritance
{
    public static void main()
    {
        X ob=new X();
    }
}
```

X() has private access in X

The constructor is private so object cannot be created.

DEFAULT ACCESS MODIFIER

```
package mypack;
class A{
    void msg(){System.out.println("Hello");}
}

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

The 2nd program shows a error because it cannot find class A. Default access modifier has no keyword.

✚ PROTECTED ACCESS MODIFIER

<pre>package pack; public class A { protected void msg() { System.out.println("Hello"); } }</pre>	<pre>package mypack; import pack.*; class B extends A { public static void main(String args[]) { B obj = new B(); obj.msg(); } }</pre>
---	---

It only works with inheritance and class A must be declared public. A class cannot be made protected.

✚ PUBLIC ACCESS MODIFIER

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

ACCESS MODIFIER AND METHOD OVERRIDING

- Parent class function is protected

```
class ABC
{
    protected void msg()
    {
        System.out.println("Hello java");
    }
}
```

```
public class Simple extends ABC
```

```
{
    void msg()
    {
        S
    }
    public static void main(String args[])
    {
        Simple obj=new Simple();
        obj.msg();
    }
}
```

msg() in Simple cannot override msg() in ABC attempting to assign weaker access privileges; was protected

- Parent class function is private

```
class ABC
{
    private void msg()
    {
        System.out.println("Hello java");
    }
}

public class Simple extends ABC
{
    void msg()
    {
        System.out.println("Hello CODER");
    }
    public static void main(String args[])
    {
        Simple obj=new Simple();
        obj.msg();
    }
}
```

- Parent class function is public

```
class ABC
{
    public void msg()
    {
        System.out.println("Hello java");
    }
}

public class Simple extends ABC
{
    void msg()
    {
        msg() in Simple cannot override msg() in ABC
        attempting to assign weaker access privileges; was
        public
    }

    public static void main(String args[])
    {
        Simple obj=new Simple();
        obj.msg();
    }
}
```

Inheritance and access modifiers

```
import java.util.*;
public class singleLevelInheritance
{
    int a;
    protected int b;
    private int c;
    public String name;
    public String uid;
    public singleLevelInheritance()
    {
        a=10;
        b=20;
        c=30;
        name="Shreyans";
        uid="18BCS3465";
    }
    public singleLevelInheritance(int a, int b)
    {
        this();
        this.a = a;
        this.b = b;
    }
    public void display()
    {
        System.out.println("Value of c:"+c);
        System.out.println(name);
        System.out.println(uid);
    }
    private void privateFunction()
    {
        System.out.println("This function is not inherited");
    }
    protected void protectedFunction()
    {
        System.out.println("This is protected function");
    }
    void defaultFunction()
    {
        System.out.println("This is default function");
    }
}
```



```
import java.util.*;
public class test extends singleLevelInheritance
{
    int c;
    public test()
    {
    }
    public test(int a,int b,int c)
    {
        super(a,b);
        this.c = c;
    }
    public static void main(String[] args)
    {
        Scanner in=new Scanner(System.in);

        System.out.println("enter value of a:");
        int a=in.nextInt();
        System.out.println("enter value of b:");
        int b=in.nextInt();

        test ob=new test(a,b,100);
        System.out.println("a="+ob.a);
        System.out.println("b="+ob.b);
        System.out.println("c="+ob.c);
        ob.protectedFunction();
        //ob.privateFunction();//not inherited
        ob.defaultFunction();
        ob.display();
    }
}
```

Packages in Java

The packages in JAVA is a mechanism to encapsulate group of classes, sub-classes or interfaces. It is the container for classes used to keep them compartmentalized.

A java package is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc. Object class is the parent of all classes in java. Java.lang.object

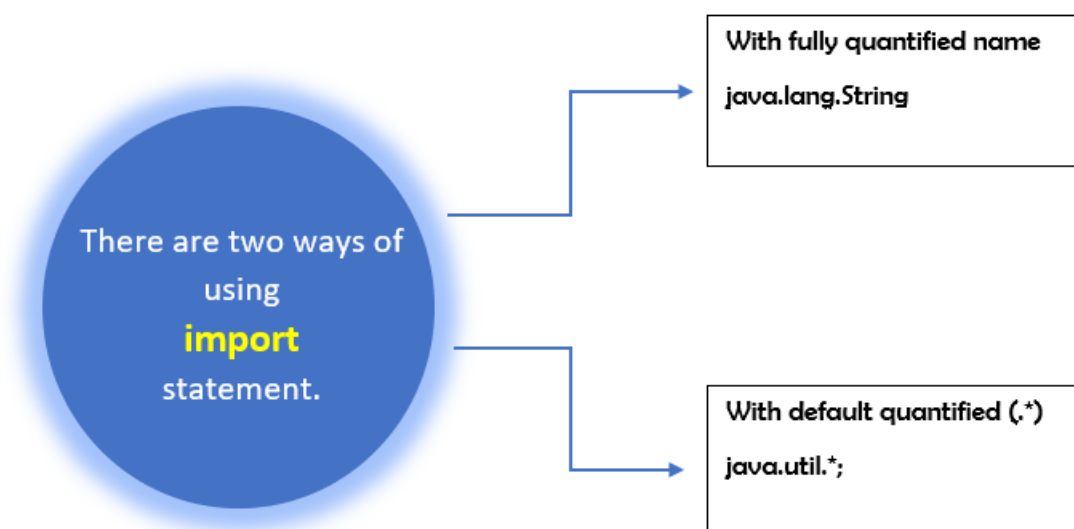
Advantage of Java Package

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.
- Packages provide reusability of code.
- To bundle classes and interfaces.
- We can create our own Package or extend already available Package.

Built in packages in java

There are various built-in packages in java. Java has lot of packages called API bundled with the JDK. Packages is used to facilitate the use of built in packages and solutions.

The user can also define their own packages.



Instead of importing the whole package it is possible to refer a class in order to instantiate the object. The two ways possible are as follows: -

<pre>class packages { public static void main() { java.util.Date toDay=new java.util.Date(); System.out.println(toDay); } }</pre>	<pre>import java.util.Date; class packages { public static void main() { Date toDay=new java.util.Date(); System.out.println(toDay); } }</pre>
--	--

The package keyword is used to create your own package. The sub-packages can also be made.

<pre>package MyPackage; //save as My class.java public class MyClass { public void test () { System.out.println (" Welcome to My Class !"); } }</pre>	<pre>import MyPackage.MyClass; class PackageTestAppin { public static void main (String args []) { MyClass theClass = new MyClass (); theClass.test (); } }</pre>
---	---

Save as myClass.java

Save as PackageTestAppin.java

To access a package in its subfolder, path should be specified. For example to access package c kept in b package we write package b.c.[class name or *] is used. * is used to import the entire package.

Non-subclass=no inheritance

Access protection for packages				
	Private	Default	Protected	Public
Same class	Yes	Yes	Yes	Yes
Sub-classes in same package	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Sub-classes in different packages	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

- Default access modifier is inaccessible outside package.
- Private access modifier inaccessible outside class.
- Protected access modifier inaccessible from non-sub class in different package. Accessible in same package non sub-class.
- Public access modifier accessible everywhere.

There are two ways to import the packages

- Implicitly by using the import keyword.
For example, `import java.util.*;`
- Explicitly by using it directly form the source.
For example, `java.util.Scanner....`

<pre>import java.util.Scanner; class Implicit_package { public static void main(String[] args) { Scanner in=new Scanner (System.in); System.out.println("Implicit declaration of package"); } }</pre>	<pre>class Explicit_package { public static void main(String[] args) { java.util.Scanner in=new java.util.Scanner (System.in); System.out.println("Explicit declaration of package"); } }</pre>
---	---

Custom packages can also be used. Creating your own package is useful when creating your own applications. It can even be used to organize your java programs.

```
package Package;

public class numbers
{
    public int square(int n)
    {
        return n*n;
    }
    public static void output()
    {
        System.out.println("Static function here");
    }
}
```

```
import Package.numbers;
import java.util.*;
class Package_access
{
    public static void main()
    {
        Scanner in=new Scanner(System.in);

        System.out.println("Enter the number");
        int n=in.nextInt();

        numbers ob=new numbers();
        System.out.println(ob.square(n));
        numbers.output();

    }
}
```

Multiple packages can also be imported or used. The class inside the package should be declared public to access it outside the package.

It is quite possible to have classes of same name inside two different packages. However, both classes cannot be imported in the same file.

To do this object can be made by using fully quantified name(explicitly).

<pre>package pack; public class abc { public void display() { System.out.println("pack package"); } }</pre>	<pre>package mypack; public class abc { public void display() { System.out.println("mypack package"); } }</pre>
<pre>class Package_Same_Class { public static void main() { pack.abc ob1=new pack.abc(); mypack.abc ob2=new mypack.abc(); ob1.display(); ob2.display(); } }</pre>	

Abstract Class in Java

```
abstract class Bike
{
    abstract void run();
}

class Honda extends Bike
{
    void run()
    {
        System.out.println("Running Safetly");
    }
    public static void main()
    {
        Bike ob=new Honda();
        ob.run();
    }
}
```

Abstract classes and Abstract methods:

- An abstract class is a class that is declared with abstract keyword.
- An abstract method is a method that is declared without an 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 either make subclass itself abstract.
- Any class that contains one or more abstract methods must also be declared with abstract keyword.
- There can be no object of an abstract class. That is, an abstract class cannot be directly instantiated with the *new operator*.
- An abstract class can have parametrized constructors and default constructor is always present in an abstract class.
- An abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of an inherited class is created.

Interfaces in Java

The Abstract class: -

- ✓ Any class with abstract method is automatically Abstract itself and need to be specified.
- ✓ A class may be declared abstract even if it does not have any abstract method.
- ✓ A sub-class of an abstract class can be instantiated if it overrides all methods of its parent class.
- ✓ If sub-class of an abstract class does not implement all of the abstract methods, it inherits, that sub-class is itself abstract.

The multiple inheritance is not possible in java. To implement the multi-inheritance in java interfaces are used. The abstract class only shows single inheritance whereas interface show multiple inheritance even though they are similar.

An interface is a kind of class, but it has all methods abstract and all variables final. Like an abstract class, it cannot be instantiated.

The abstract might contain non-abstract methods but interface cannot. Interface gives the protocol of behavior that can be implemented by any class.

An interface defines the set of methods but does not implement them. It is a named collection of method definitions (without implementation). The class that implements the interface agrees to implement all methods defined in the interface, thereby interface reserve behavior for classes that implement them.

- Methods declared in interface are public and abstract.
- Static methods cannot be declared in the interface. A static method do not express behavior of objects.
- All variables defined in interface is public, static and final.
- The interface methods cannot be final as they are abstract.
- The interface may extend other interfaces but cannot implement other interface or class.
-

An abstract class can have the static method.

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

```
interface inter
{
    int n=13;
    void output();
}
```


The keywords `public` and `abstract` may not be used as the compiler implicitly takes them to be `public` and `abstract`.

Multiple classes can be implemented from an interface. A class can implement number of interfaces.

<pre>interface inter { int n=13; void output(); }</pre>	<pre>import java.util.*; class testInterface implements inter { public void output() { System.out.println("Hello coders "+n); } public static void main() { testInterface ob=new testInterface(); ob.output(); inter object; object=ob; object.output(); } }</pre>
---	---

Interface can inherit from other interfaces. Interface can multiply inherits.

<pre>interface Constants { }</pre>	<pre>interface Chemistry extends Constants { }</pre>
<pre>interface Physics { }</pre>	<pre>interface Chemistry extends Constants,Physics { }</pre>

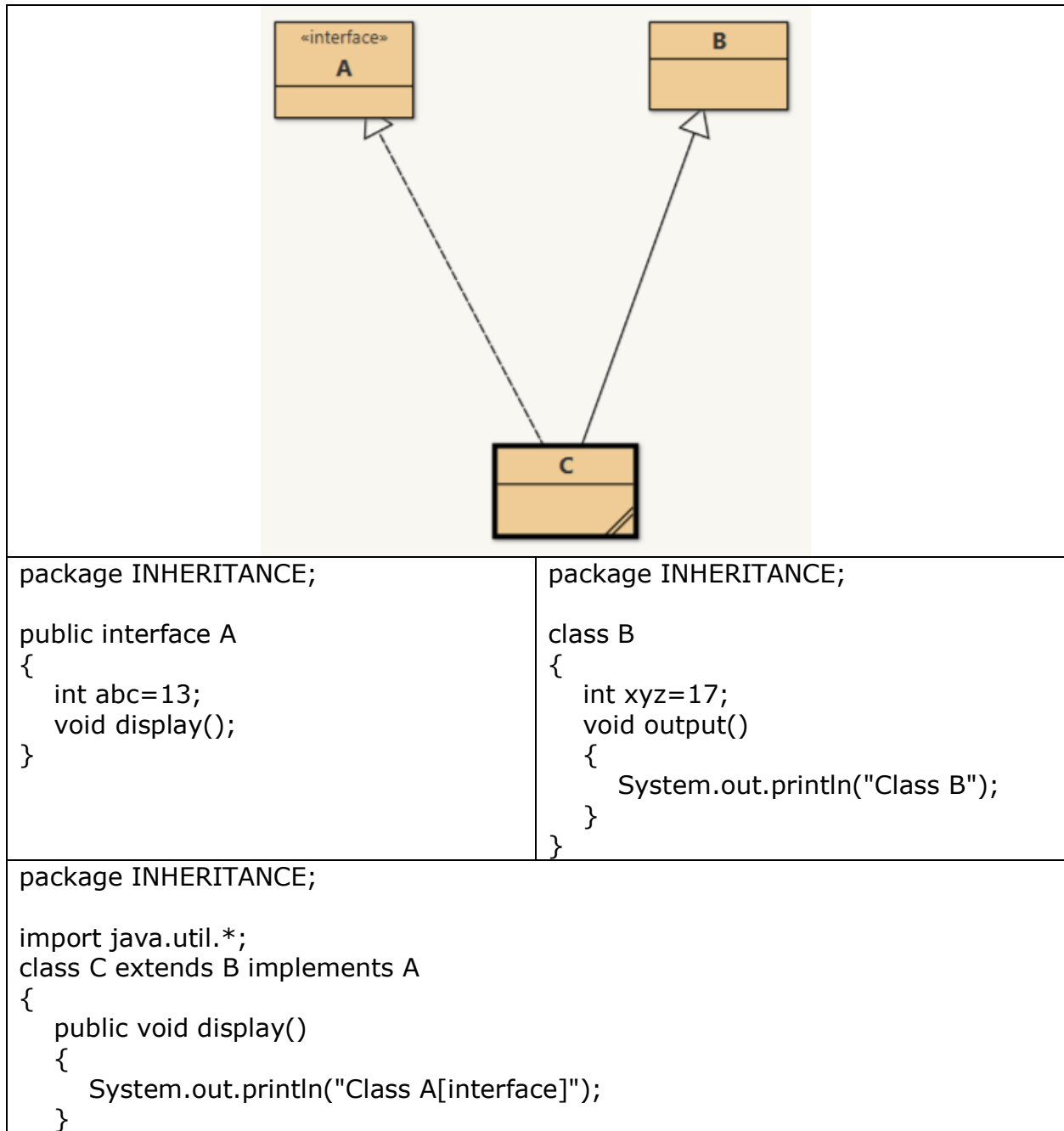
DIFFERENCE BETWEEN INTERFACE AND ABSTRACT CLASS		
Difference	Interface	Abstract class
Type of Methods	Can only have abstract methods.	Can have both abstract and non-abstract methods.
Variables	The keywords declared in interface are final and static by default.	It may contain non-final or non-static variables also.
Implementation	It cannot provide implementation of abstract class.	It can provide implementation of interface.
Keyword	Implements	Extends
Multiple implementation	An interface can extend another Java interface only.	An abstract class can extend another Java class and implement multiple Java interfaces.
Access Specifier	It only have public members	It can have members like private, protected etc.

Inheritance using Interfaces

There are two types of inheritance which is possible using interfaces.

- Multilevel inheritance
- Hybrid inheritance (some parts)

Multilevel inheritance



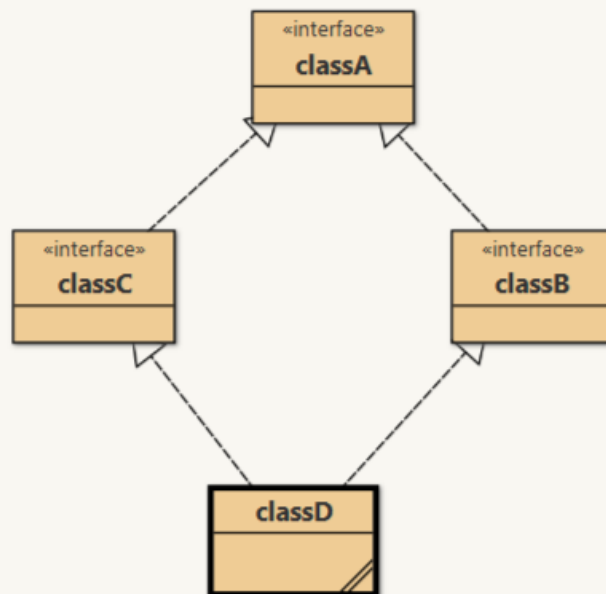
```

public static void main()
{
    Scanner in=new Scanner(System.in);

    C ob=new C();
    ob.display();
    ob.output();
    System.out.println("value of abc="+ob.abc);
    System.out.println("value of xyz="+ob.xyz);
}
}

```

Hybrid inheritance



```
package INHERITANCE;
```

```

public interface classA
{
    void out();
}

```

```
package INHERITANCE;
```

```

public interface classC extends classA
{
    void outpu();
}

```

```
package INHERITANCE;
```

```

public interface classB extends classA
{
    void outpt();
}

```

```
package INHERITANCE;
```

```
import java.util.*;

class classD implements classB,classC
{
    public void out()
    {
        System.out.println("Class A");
    }
    public void outpt()
    {
        System.out.println("class B");
    }
    public void outpu()
    {
        System.out.println("class C");
    }
    public static void main()
    {
        Scanner in=new Scanner(System.in);

        classD obj=new classD();
        obj.out();
        obj.outpu();
        obj.outpt();
    }
}
```

The package used here is not required. It just states that all inheritance programs are kept at the same place.

Exception Handling

Exception- A exception is a unexpected event that occurs during the execution of program. It disrupts the normal flow of program.

Java provides Runtime Error Management System to deal with errors and exception. All exception and errors types are sub classes of class **Throwable**, which is base class of hierarchy.

In Java, whenever an error occurs inside the java it creates the object of respective Exception class and passes it to JVM. Exception object contains name and description of the exception, and current state of the program where exception has occurred. Creating the Exception Object and handling it to the run-time system is called throwing an Exception.

Exception handling is done by 5 keywords:

- I. try{.....}
- II. catch{.....}
- III. throw
- IV. throws
- V. finally{.....}

The exception class is provided in java.lang package. The option to create their own exception makes java more flexible.

All java exception must be a part of class throwable. The super class of all errors and exceptions is throwable class. It is defined in java.lang package. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.

Exception handling in java

I. Single try-catch block

```
import java.util.*;

class tryCatch1
{
    public static int divideByZero(int a,int b)
    {
        return a/b;
    }
    public static void main()
    {
        try
        {
            System.out.println(tryCatch1.divideByZero(13,7));
            System.out.println(tryCatch1.divideByZero(13,0));
        }
        catch(ArithmeticException e)
        {
            System.out.println("Divide by zero error");
        }
    }
}
```

II. Try with multiple catch

```
import java.util.*;

class tryCatch2
{
    public static int divideByZero(int a,int b)
    {
        return a/b;
    }
    public static void main()
    {
        try
        {
            int a[]=new int[0];
            System.out.println(a[12]);
            System.out.println(tryCatch2.divideByZero(13,0));
        }
        catch(ArithmeticException e)
        {
            System.out.println("Divide by zero error");
        }
    }
}
```

```

    }
    catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println("Array index out of bounds");
    }
}

```

III. Multiple exceptions (with one catch)

```

import java.util.*;

class tryCatch3
{
    public static void main()
    {
        for(int i=0;i<4;i++)
        {
            try
            {
                switch(i)
                {
                    case 1 :  int b[]={ };  int j=b[0];
                    case 2 :  int a[]=new int[0];    System.out.println(a[12]);
break;
                    case 3 :  System.out.println(13/0);  break;
                }
            }
            catch(Exception e)
            {
                System.out.println(e.getMessage());
            }
        }
    }
}

```

IV. Exception with exit code (with finally)

```

import java.util.*;

class tryCatch4
{
    public static void main()
    {
        try
        {
            System.out.println(13/0);

```



```

        int a[]=new int[0];
        System.out.println(a[12]);
    }
    catch(ArithmeticException e)
    {
        System.out.println("Divide by zero error");
    }
    catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println("Array index out of bounds");
    }
    finally
    {
        System.out.println("Executes if exception is caught or not");
    }
}
}

```

V. Throwing an own exception (throw in try block)

To throw an own exception concept of inheritance is used to create a MyException class with a constructor. The keyword 'Super' is used to call constructor of parent class Exception with a parameter.

<pre> class MyException extends Exception { MyException(String message) { super(message); } } </pre>	<pre> import java.util.*; class tryCatch5 { public static void main() { int a=13; try { if(a==13) throw new MyException("Exception Message"); } catch(MyException e) { System.out.println(e.getMessage()); } } } </pre>
--	---

VI. Nested try-catch block

```

import java.util.*;

class tryCatch6
{
    public static int divideByZero(int a,int b)
    {
        return a/b;
    }
    public static void tryInsideTryBlock()
    {
        try
        {
            System.out.println(tryCatch6.divideByZero(13,0));
        }
        catch(ArithmeticException e)
        {
            System.out.println("Inside Exception 'Divide by zero error'");
        }
    }
    public static void main()
    {
        try
        {
            tryCatch6.tryInsideTryBlock();
            tryCatch6.divideByZero(17,0);
        }
        catch(Exception e)
        {
            System.out.println("Outside Exception '"+e.getMessage()+"'");
        }
    }
}

```

VII. Throwing an own exception (throws in try block)

The keyword 'throws' is used to throw a multiple exceptions which may occur in the program. In the input program, an Input Output exception is thrown.

```

import java.io.*;

class Data_Input_Stream
{
    public static void main()throws IOException
    {
        DataInputStream in=new DataInputStream(System.in);

        System.out.println("Enter a number");
    }
}

```

```
int n;  
n=Integer.parseInt(in.readLine());  
  
System.out.printf("You entered:%d",n);  
  
}  
}
```

There are many ways to use throws keyword. In the above program it can be used after input of n also.