

ENCAPSULATION AND ABSTRACTION

ENCAPSULATION

>Encapsulation in Java is a process of wrapping code and data together into a single unit

>We can create a fully encapsulated class in Java by making all the data members of the class private and use setter n getter methods to set n get the data in it.

>It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.

>The Java Bean class is the example of a fully encapsulated class.

>It provides you the control over the data.

>The encapsulate class is easy to test. So, it is better for unit testing.

>In encapsulation, the data is hidden using methods of getters and setters.

>In this, problems are solved at the implementation level.

>encapsulation can be implemented using by access modifier i.e. private, protected and public.

Ex:

```
public class Student
```

```
{
```

```
    //private data member
```

```
    private String name;
```

```
    //getter method for name
```

```
    public String getName()
```

```
{
```

```
    return name;
```

```

}

//setter method for name

public void setName(String name){
    this.name=name
}

}

class Test

{

    public static void main(String[] args)
    {

        //creating instance of the encapsulated class
        Student s=new Student();

        //setting value in the name member
        s.setName("nani");

        //getting value of the name member
        System.out.println(s.getName()); //prints vijay
        System.out.println(s.name);      //Compile Time Error,
    }

}

```

>By providing only a setter or getter method, you can make the class read-only or write-only

Ex:

READ-ONLY

```

public class Student{
    //private data member
    private String college="AKG";
}

```

```
//getter method for college
public String getCollege()
{
    return college;
}

}
s.setCollege("KITE");//will render compile time error
```

WRITE-ONLY

```
public class Student{
    //private data member
    private String college;
    //getter method for college
    public void setCollege(String college)
    {
        this.college=college;
    }
}
System.out.println(s.getCollege());//Compile Time Error, because there is no such method
```

ABSTRACTION

>Data Abstraction is defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details

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

>Abstract class cannot be instantiated using new() operator.

>Abstraction lets you focus on what the object does instead of how it does it.

>In abstraction, implementation complexities are hidden using abstract classes and interfaces.

>In abstraction, problems are solved at the design or interface level.

>Abstraction can be implemented using

i)Abstract class

ii)Interface

Abstract class

>A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods.

>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

>If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

>Any class that contains one or more abstract methods must also be declared with abstract keyword otherwise throws compile time error.

>It can have constructors and static methods also.

>It can have final methods which will force the subclass not to change the body of the method.

Ex:

```
abstract class Shape
```

```
{
```

```
    Shape()
```

```
{
```

```
    System.out.println("Shape constructor called");
```

```
}
```

```
    abstract void draw();
```

```
}
```

```

//In real scenario, implementation is provided by others i.e. unknown by end user

class Rectangle extends Shape
{
    void draw()
    {
        System.out.println("drawing rectangle");
    }
}

class Circle1 extends Shape
{
    void draw()
    {
        System.out.println("drawing circle");
    }
}

//In real scenario, method is called by programmer or user

class TestAbstraction1
{
    public static void main(String args[])
    {
        Shape s=new Circle1();
        s.draw();
    }
}

```

O/P:

Shape constructor called

drawing circle

>If there is an abstract method in a class, that class must be abstract.

Ex:

class Bike

{

 abstract void run();

}

O/P:

compile time error

ARRAYS

=====

>Array is an indexed collection of fixed number of homogenous data elements.

>Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value which improves readability.

>Every array is an object in java as it is created using new keyword.

>The direct superclass of an array type is Object.

>Every array type implements the interfaces Cloneable and java.io.Serializable.

>Main advantages of using arrays are easy to index, search and sort.

Features of Array

>Arrays are objects

- >They can even hold the reference variables of other objects
- >They are created during runtime
- >They are dynamic, created on the heap
- >The Array length is fixed

Advantages

- >collection of similar types of data.
- >Easy to memorize.
- >Used to implement other data structure like linked lists, stack, queue, trees, graph etc.
- >2 Dimensional array is used to represent a matrix.

DISADVANTAGES

- >Time complexity increase in insertion and deletion operation.
- >wastage of memory because arrays are fixed in size.
- >If there is enough space present in the memory bt not in contiguous form , in this case you will not able initialize your array.
- >It is not possible to increase the size of the array, once you had declared the array.

One-dimensional array declaration:

Three ways to declare an array

```
int[] x; (recommended)  
int []x;  
int x[];
```

Ex:

`int[6] x;` throws CE bcz at the time of declaration we cant specify the size.

```
int[] x;
```

Two-dimensional array declaration:

6 ways to declare an 2D array

```
int[][] x;
```

```
int [][]x;
```

```
int x[][];
```

```
int[] []x;
```

```
int[] x[];
```

```
int []x[];
```

Ex:

int[] []a,[]b; gives CE bcz if we want to specify before the variable, it is applicable only for first variable.

Three dimensional array declaration:

Ways to declare 3Darray:

```
int[][][] x;
```

```
int [][][]x;
```

```
int x[][][];
```

```
int[] [][]x;
```

```
int[] x[][];
```

```
int[][] a[];
```

```
int[][] []a;
```

```
int [][]a[];
```

```
int []a[][];
```

Array creation/instantiation:

- > Every array is an object in java.
- > While creating, we need to provide size for an array bcz array is stored in heap memory basd on that size.

```
int[] a =new int[5];
```

- > We can't provide negative array size

```
int[] a= new int[-2];
```

negativeArraySizeException at run-time only.

- > To specify array size, allowed data types are byte,short,int,char.
- > If we are trying to specify any other types then we wil get CE.
- > An array can contain objects of a class depending on the def of array.

>An array can also hold object references.

Ex:

```
Employee emp[]=new Employee[3];
```

The above statement creates only an array of 3 elements which holds 3 employee references emp[0], emp[1], emp[2].

**Note that the employee objects are not yet created. Referring to employee objects leads to Runtime Exception.

The following code initializes the employee objects

```
for(int i=0;i<emp.length;i++)  
{  
    emp[i]=new Employee();  
}
```

Ex:

```
public class Main
{
    public static void main(String[] args) {
        System.out.println("Hello World");
        int[] a=new int[3];
        System.out.println(a.getClass().getName());      //[I

        int[] x=new int[-6];                         //negativearraysize
        int[] x1=new int['a'];
        int[] x2=new int[2147483641];                //java heap space error
        byte[] x3=new byte[4];

        System.out.println(x1.length);
        System.out.println(args.length);
        System.out.println(a);                      //prints address of an array
    }
}
```

Array initialization:

>In a situation, where the size of the array and variables of array are already known, array literals can be used.

Ex:

--

```
int[] myarr={1,2,3,4};
```

>Another way to initialize an array.

```
int[] myarr=new int[4];
```

```
myarr[0]=1;  
myarr[1]=2;  
myarr[2]=3;  
myarr[3]=4;
```

We can declare, instantiate and initialize the java array together by:

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

Accessing array elements:

>We can access through for,foreach etc.

```
import java.util.Arrays;  
  
public class Main  
{  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
        int[] myarr=new int[4];  
        myarr[0]=1;  
        myarr[1]=2;  
        myarr[2]=3;  
        myarr[3]=4;  
        for(int i=0;i<myarr.length;i++)  
        {  
            System.out.println(myarr[i]);  
        }  
    }  
}
```

```

for(int i:myarr)
{
    System.out.println(i);
}

System.out.println("toString");
System.out.println(Arrays.toString(myarr));
}

}

myarr=null;
System.out.println(myarr[2]);

```

The above statement makes the array to point to null, means the array object which was on the heap is ready for garbage collection.

Referring to array object now gives a NullPointerException.

Anonymous array in java:

>An array in Java without any name is anonymous array. It is an array just for creating and using instantly.

>Anonymous array is passed as an argument of method

Syntax:

```
// anonymous int array
new int[] { 1, 2, 3, 4};
```

```
// anonymous char array  
new char[] {'x', 'y', 'z'};  
  
// anonymous String array  
new String[] {"Geeks", "for", "Geeks"};  
  
// anonymous multidimensional array  
new int[][] { {10, 20}, {30, 40, 50} };
```

Ex:

```
// Java program to illustrate the  
// concept of anonymous array  
class Test {  
    public static void main(String[] args)  
    {  
        // anonymous array  
        sum(new int[]{ 1, 2, 3 });  
  
    }  
    public static void sum(int[] a)  
    {  
        int total = 0;  
  
        // using for-each loop  
        for (int i : a)  
            total = total + i;  
  
        System.out.println("The sum is:" + total);  
    }  
}
```

```
 }  
 }
```

Passing array to a method:

>We can pass array as an argument for a method.

```
public class Main  
{  
    public static void main(String[] args)  
    {  
        // anonymous array  
        int a[]={1,2,3};  
        sum(a);  
        //      sum(new int[]{ 1, 2, 3 });  
    }  
    public static void sum(int[] a)  
    {  
        int total = 0;  
  
        // using for-each loop  
        for (int i : a)  
            total = total + i;  
  
        System.out.println("The sum is:" + total); //sum is 6  
    }  
}
```

Array get() Method in Java

>It is an inbuilt method in Java and is used to return the element at a given index from the specified Array.

Syntax:

Array.get(Object []array, int index)

array: The object array whose index is to be returned.

index: The particular index of the given array. The element at 'index' in the given array is returned

>This method returns the element of the array as type of Object class.

>Type casting is compulsory for returning an element bcz by default it is an object class so we need to convert it into programmer defined element type.

Ex:

import java.lang.reflect.Array;

```
public class GfG {  
    // main method  
    public static void main(String[] args)  
    {  
        // Declaring and defining an int array  
        int a[] = { 1, 2, 3, 4, 5 };  
  
        // Array.get method  
        // Note : typecasting is essential  
        // as the return type is Object.  
        int x = (int)Array.get(a, 3);
```

```
// Printing the value
System.out.print(x + " ");
//4
}
}
```

Returning array from method

>We can also return an array from the method in Java.

//Java Program to return an array from the method

```
class TestReturnArray
{
    //creating method which returns an array
    static int[] values()
    {
        return new int[]{10,30,50,90,60};
    }

    public static void main(String args[])
    {
        //calling method which returns an array
        int arr[]=values();
        //printing the values of an array
        for(int i=0;i<arr.length;i++)
            System.out.println(arr[i]);
    }
}
```

What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

A Collection is a group of individual objects represented as a single unit. Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.

What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main “root” interfaces of Java collection classes.

Need for Collection Framework :

Before Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors or Hashtables. All of these collections had no common interface.

Accessing elements of these Data Structures was a hassle as each had a different method (and syntax) for accessing its members:

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

class Test
{
    public static void main (String[] args)
    {
        // Creating instances of array, vector and hashtable
        int arr[] = new int[] {1, 2, 3, 4};
        Vector<Integer> v = new Vector();
        Hashtable<Integer, String> h = new Hashtable();
        v.addElement(1);
        v.addElement(2);
        h.put(1,"geeks");
        h.put(2,"4geeks");

        // Array instance creation requires [], while Vector
        // and hashtable require ()
    }
}
```

```

// Vector element insertion requires addElement(), but
// hashtable element insertion requires put()

// Accessing first element of array, vector and hashtable
System.out.println(arr[0]);
System.out.println(v.elementAt(0));
System.out.println(h.get(1));

// Array elements are accessed using [], vector elements
// using elementAt() and hashtable elements using get()
}
}

```

As we can see, none of these collections (Array, Vector or Hashtable) implement a standard member access interface. It was very difficult for programmers to write algorithms that can work for all kinds of Collections. Another drawback being that most of the 'Vector' methods are final, meaning we cannot extend the 'Vector' class to implement a similar kind of Collection.

REFERENCE LINK:

<https://www.geeksforgeeks.org/collections-in-java-2/>

Sr.No.	Method & Description
1	boolean add(Object obj) Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.
2	boolean addAll(Collection c) Adds all the elements of c to the invoking collection. Returns true if the operation succeeds (i.e., the elements were added). Otherwise, returns false.
3	void clear()

	Removes all elements from the invoking collection.
4	boolean contains(Object obj) Returns true if obj is an element of the invoking collection. Otherwise, returns false.
5	boolean containsAll(Collection c) Returns true if the invoking collection contains all elements of c. Otherwise, returns false.
6	boolean equals(Object obj) Returns true if the invoking collection and obj are equal. Otherwise, returns false.
7	int hashCode() Returns the hash code for the invoking collection.
8	boolean isEmpty() Returns true if the invoking collection is empty. Otherwise, returns false.
9	Iterator iterator() Returns an iterator for the invoking collection.
10	boolean remove(Object obj) Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.
11	boolean removeAll(Collection c) Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.
12	boolean retainAll(Collection c)

	Removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.
13	int size() Returns the number of elements held in the invoking collection.
14	Object[] toArray() Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
15	Object[] toArray(Object array[]) Returns an array containing only those collection elements whose type matches that of array.

REFERENCE LINK:

https://www.tutorialspoint.com/java/java_collection_interface.htm

add():

The **add(E element)** of **java.util.Collection interface** is used to add the element 'element' to this collection. This method returns a boolean value depicting the successfulness of the operation. If the element was added, it returns true, else it returns false.

Syntax:

Collection.add(E element)

Parameters: This method accepts a mandatory parameter **element** of type E which is to be added to this collection.

Return Value: This method returns a **boolean value** depicting the successfulness of the operation. If the element was added, it returns true, else it returns false.

Exceptions: This method throws following exceptions:

- **UnsupportedOperationException:** if the add operation is not supported by this collection
- **ClassCastException:** if the class of the specified element prevents it from being added to this collection

- **NullPointerException**: if the specified element is null and this collection does not permit null elements
- **IllegalArgumentException**: if some property of the element prevents it from being added to this collection
- **IllegalStateException**: if the element cannot be added at this time due to insertion restrictions

```

import java.io.*;
import java.util.*;
public class Main {
    public static void main(String args[])
    {
        // creating an empty LinkedList
        Collection<String> list = new LinkedList<String>();
        // use add() method to add elements in the list
        list.add("Geeks");
        list.add("for");
        list.add("Geeks");
        Collection<String> de_que = new ArrayDeque<String>();
        // Use add() method to add elements into the Deque
        de_que.add("Welcome");
        de_que.add("To");
        de_que.add("Geeks");
        Collection<Integer> arrlist = new ArrayList<Integer>(5);
        // use add() method to add elements in the list
        arrlist.add(15);
        arrlist.add(20);
        arrlist.add(25);
    }
}

```

```
// printing the new list  
System.out.println("The new List is: " + list);  
System.out.println("ArrayDeque: " + de_QUE);  
System.out.println("ArrayList :" + arrlist);  
}  
}
```

Output:

```
The new List is: [Geeks, for, Geeks]
```

```
ArrayDeque: [Welcome, To, Geeks]
```

```
ArrayList :[15, 20, 25]
```

addAll();

The **addAll()** method of **java.util.Collections** class is used to add all of the specified elements to the specified collection. Elements to be added may be specified individually or as an array.

```
/*//DEMO OF addAll()  
import java.util.*;  
import java.io.*;  
public class Main {  
    public static void main(String[] argv) throws Exception  
    {  
  
        // creating object of List<String>  
        List<String> arrlist = new ArrayList<String>();  
  
        // Adding element to arrlist  
        arrlist.add("A");
```

```
arrlist.add("B");
arrlist.add("C");
arrlist.add("Tajmahal");

// printing the arrlist before operation
System.out.println("arrlist before operation : " + arrlist);

// add the specified element to specified Collections
// using addAll() method
boolean b = Collections.addAll(arrlist, "1", "2", "3");
// printing the arrlist after operation
System.out.println("result : " + b);

// printing the arrlist after operation
System.out.println("arrlist after operation : " + arrlist);
}

}/*
// ADDING ELEMENTS IN LINKEDLIST DEMO
import java.util.*;
import java.io.*;
public class Main {
    public static void main(String args[])
    {

        // Creating an empty LinkedList
        Collection<String>
        list = new LinkedList<String>();
```

```

// A collection is created
Collection<String>
    collect = new LinkedList<String>();
    collect.add("A");
    collect.add("Computer");
    collect.add("Portal");
    collect.add("for");
    collect.add("Geeks");

// Displaying the list
System.out.println("The LinkedList is: " + list);

// Appending the collection to the list
System.out.println(list.addAll(collect));
System.out.println("Elements of collect linkedList"+collect);
System.out.println("The new linked list is:"
    + list);
}

}

Output:

```

```

The LinkedList is: []
true
Elements of collect linkedList[A, Computer, Portal, for, Geeks]
The new linked list is: [A, Computer, Portal, for, Geeks]

```

REFERENCE LINK:

<https://www.javatpoint.com/java-collections-addall-method>

remove():

The remove() method of Java Collection Interface is used to remove a single instance of the specified element from this collection.

```
1. import java.util.Collection;
2. import java.util.HashSet;
public class Main {

    public static void main(String[] args) {

        Collection<Integer> collection = new HashSet<>();
        collection.add(5);
        collection.add(15);
        collection.add(52);
        collection.add(532);
        collection.add(52);
        System.out.println(collection);
        //will remove the specified element from the collection
        Boolean b= collection.remove(52);
        System.out.println("After removing 52 \nCollection : "+collection);
        System.out.println(b);
    }
}
```

Output:

```
[52, 532, 5, 15]
```

```
After removing 52
```

```
Collection : [532, 5, 15]
```

1. **public class** JavaCollectionRemoveExample3 {
2. **public static void** main(String[] args) {

```

3. Collection<String> collection = new ArrayDeque<String>() {
4. };
5. collection.add("Reema");
6. collection.add("Geetanjali");
7. //pass an exception for null elements
8. collection.add(null);
9.
10.    for (String i:collection) {
11.        System.out.println(i);
12.    }
13.    //will remove the specified element from the collection
14.    boolean val=collection.remove("ABC");
15.    System.out.println("Remove method will return : "+val);
16. }
17. }
```

output:

```

Exception in thread "main" java.lang.NullPointerException
at java.util.ArrayDeque.addLast(ArrayDeque.java:249)
at java.util.ArrayDeque.add(ArrayDeque.java:423)
at
com.javaTpoint.JavaCollectionRemoveExample3.main(JavaCollectionRemoveExample3.java:13)
```

REFERENCE LINK:

<https://www.javatpoint.com/java-collection-remove-method>

contains():

The **contains(Object element)** of **java.util.Collection interface** is used to check whether the element 'element' exists in this collection. This method returns a boolean value depicting the presence of the element. If the element is present, it returns true, else it returns false.

```

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

        // creating an empty LinkedList
        Collection<String> list = new LinkedList<String>();

        // use add() method to add elements in the list
```

```

list.add("Geeks");
list.add("for");
list.add("Geeks");

// Output the present list
System.out.println("The list is: " + list);

// Checking the presence of Geeks
// using contains() method
boolean result = list.contains("Geeks");

// printing the result
System.out.println("Is Geeks present in the List: "
+ result);
}
}

```

Output:

```

The list is: [Geeks, for, Geeks]
Is Geeks present in the List: true

```

REFERENCE LINK:

<https://www.javatpoint.com/java-collection-contains-method>

<https://www.geeksforgeeks.org/collection-contains-method-in-java-with-examples/>

containsAll():

The containsAll() method of List interface in Java is used to check if this List contains all of the elements in the specified Collection. So basically it is used to check if a List contains a set of elements or not.

Syntax:

```
boolean containsAll(Collection col)
```

Parameters: This method accepts a mandatory parameter **col** which is of the type of collection. This is the collection whose elements are needed to be checked if it is present in the List or not.

```

public class ListDemo {
    public static void main(String args[])
    {
        // Creating an empty list

```

```

List<String> list = new ArrayList<String>();

// Use add() method to add elements
// into the List
list.add("Welcome");
list.add("To");
list.add("Geeks");
list.add("4");
list.add("Geeks");

// Displaying the List
System.out.println("List: " + list);

// Creating another empty List
List<String> listTemp = new ArrayList<String>();

listTemp.add("Geeks");
listTemp.add("4");
listTemp.add("Geeks");

System.out.println("Are all the contents equal? "
+ list.containsAll(listTemp));
}
}

```

REFERENCE LINK:

<https://www.geeksforgeeks.org/list-containsall-method-in-java-with-examples/>

removeAll()

The **removeAll()** method of **java.util.ArrayList** class is used to remove from this list all of its elements that are contained in the specified collection.

Syntax:

```
public boolean removeAll(Collection c)
```

Parameters: This method takes **collection c** as a parameter containing elements to be removed from this list.

Returns Value: This method returns **true** if this list changed as a result of the call.

```
import java.util.*;
public class GFG1 {
    public static void main(String[] argv) throws Exception
    {
        ArrayList<Integer> arrlist1 = new ArrayList<Integer>();
        arrlist1.add(1);
        arrlist1.add(2);
        arrlist1.add(3);
        arrlist1.add(4);

        System.out.println("ArrayList before "
            + "removeAll() operation : "
            + arrlist1);

        ArrayList<Integer> arrlist2 = new ArrayList<Integer>();
        arrlist2.add(1);
        arrlist2.add(2);

        System.out.println("Collection Elements + to be removed : "
            + arrlist2);

        arrlist1.removeAll(arrlist2)

        System.out.println("ArrayList after "
            + "removeAll() operation : "
            + arrlist1);
    }
}
```

Output:

ArrayList before removeAll() operation : [1, 2, 3]

Collection Elements to be removed : [1, 2]

ArrayList after removeAll() operation : [3,4]

REFERENC LINK:

<https://www.geeksforgeeks.org/arraylist-removeall-method-in-java-with-examples/>

retainAll()

The **retainAll()** method of **ArrayList** is used to remove all the array list's elements that are not contained in the specified collection or retains all matching elements in the current ArrayList instance that match all elements from the Collection list passed as a parameter to the method.

Syntax:

```
public boolean retainAll(Collection C)
```

Parameters: The C is the collection containing elements to be retained in the list.

Return Value: The method returns a boolean value true if the list is changed at all as a result of the call else false.

Exceptions:

1. ClassCastException: If the class of an element of this ArrayList is incompatible with the Passed collection. This is optional.
2. NullPointerException: If the list contains a null element and the passed collection does not permit null elements, or if the specified collection is null. This is also optional.

```
import java.util.ArrayList;
public class GFG {
    public static void main(String[] args)
    {
        // Creating an empty array list
        ArrayList<String> bags = new ArrayList<String>();

        // Add values in the bags list.
        bags.add("pen");
        bags.add("pencil");
        bags.add("paper");

        // Creating another array list
        ArrayList<String> boxes = new ArrayList<String>();

        // Add values in the boxes list.
        boxes.add("pen");
```

```

        boxes.add("paper");
        boxes.add("books");
        boxes.add("rubber");

        // Before Applying method print both lists
        System.out.println("Bags Contains :" + bags);
        System.out.println("Boxes Contains :" + boxes);

        // Apply retainAll() method to boxes passing bags as parameter
        boxes.retainAll(bags);

        // Displaying both the lists after operation
        System.out.println("\nAfter Applying retainAll()"+
        " method to Boxes\n");
        System.out.println("Bags Contains :" + bags);
        System.out.println("Boxes Contains :" + boxes);
    }
}

```

Output:

```

Bags Contains :[pen, pencil, paper]
Boxes Contains :[pen, paper, books, rubber]

```

After Applying retainAll() method to Boxes

```

Bags Contains :[pen, pencil, paper]
Boxes Contains :[pen, paper]

```

REFERENCE LINK:

<https://www.geeksforgeeks.org/arraylist-retainall-method-in-java/>

toArray() Method

The toArray() method of Collection Interface returns an array containing all the elements present in the collection.

The second syntax returns an array containing all of the elements in this collection where the runtime type of the returned array is that of the specified array.

Syntax

1. **public** Object[] toArray()

```
2. public<T> T[] toArray(T[] a)
3. import java.util.Collection;
4. import java.util.concurrent.ConcurrentLinkedQueue;
5. public class JavaCollectcionToArrayExample2 {
6. public static void main(String[] args) {
7.     Collection<Integer> queue = new ConcurrentLinkedQueue();
8.     for (int i=1;i<=10;i++) {
9.         queue.add(i);
10.    }
11.    Integer [] a = queue.toArray();
12.    for (int i = 0; i <a.length; i++) {
13.        System.out.print(a[i] + " ");
14.        if (a[i] %2==0) {
15.            System.out.println(a[i]+ " is an even number.");
16.        }
17.    }
18. }
19. }
```

Output:

List of even numbers in our collection.

```
2
4
6
8
10
```

EXAMPLE WITH EXCEPTION:

```
1. import java.util.Collection;
2. import java.util.concurrent.ConcurrentLinkedQueue;
3. public class JavaCollectcionToArrayExample2 {
4. public static void main(String[] args) {
5.     Collection<Integer> queue = new ConcurrentLinkedQueue();
6.     for (int i=1;i<=10;i++) {
7.         queue.add(i);
8.     }
9.     Object[] a = queue.toArray();
10.    for (int i = 0; i <a.length; i++) {
11.        System.out.print(a[i] + " ");
```

```

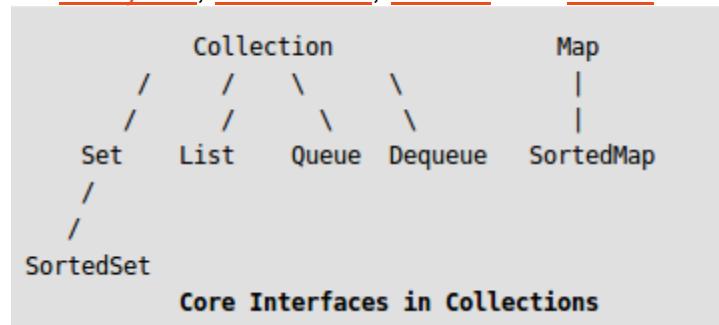
12.     if (a[i] %2==0) {
13.         System.out.println(a[i]+" is an even number.");
14.     }
15. }
16. }
17. }
```

Output:

Error:(15, 18) java: bad operand types for binary operator '%'
 first type: java.lang.Object
 second type: int

List Interface

The Java.util.List is a child interface of [Collection](#). It is an ordered collection of objects in which duplicate values can be stored. Since List preserves the insertion order, it allows positional access and insertion of elements. List Interface is implemented by the classes .of [ArrayList](#), [LinkedList](#), [Vector](#) and [Stack](#).



Declaration:

```
public abstract interface List extends Collection
```

Creating List Objects:

List is an interface, and the instances of List can be created by implementing various classes in the following ways:

```
List a = new ArrayList();
```

```
List b = new LinkedList();
```

```
List c = new Vector();
```

```
List d = new Stack();
```

Generic List Object:

After the introduction of Generics in Java 1.5, it is possible to restrict the type of

object that can be stored in the List. The type-safe List can be defined in the following way:

```
// Obj is the type of object to be stored in List  
List<Obj> list = new ArrayList<Obj>();
```

Reference Link:

<https://www.geeksforgeeks.org/list-interface-java-examples/>

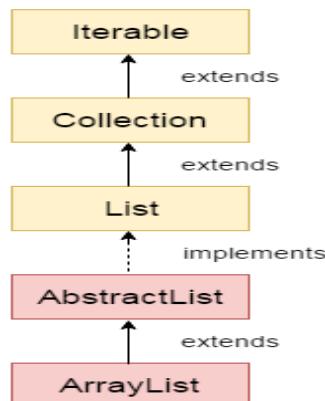
ARRAYLIST:

The **ArrayList** class is a resizable array, which can be found in the **java.util** package.

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to occur if any element is removed from the array list.



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:

Example

Create an **ArrayList** object called **cars** that will store strings:

```
import java.util.ArrayList; // import the ArrayList class
```

```
ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

Constructors of Java ArrayList

ArrayList()	built an empty array list.
ArrayList(Collection<? extends E>c)	built an array list that is initialized with the elements of the collection c
ArrayList(int capacity)	built array list that has the specified initial capacity.

Add Items

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

Example

```
import java.util.ArrayList;

public class MyClass {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
```

```
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
}
}
```

Access an Item

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

Example

```
cars.get(0);
```

Remember: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Item

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

Example

```
cars.set(0, "Opel");
```

Remove an Item

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

Example

```
cars.remove(0);
```

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

Example

```
cars.clear();
```

ArrayList Size

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

Example

```
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:

Example

```
public class MyClass {  
    public static void main(String[] args) {  
        ArrayList<String> cars = new ArrayList<String>();  
        cars.add("Volvo");  
        cars.add("BMW");  
        cars.add("Ford");  
        cars.add("Mazda");  
        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:

```
public class MyClass {  
    public static void main(String[] args) {  
        ArrayList<String> cars = new ArrayList<String>();  
        cars.add("Volvo");  
        cars.add("BMW");  
        cars.add("Ford");  
        cars.add("Mazda");  
        for (String i : cars) {  
            System.out.println(i);  
        }  
    }  
}
```

Other Types

Elements in an `ArrayList` are actually objects. In the examples above, we created elements (objects) of type "String". 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:

Example

Create an `ArrayList` to store numbers (add elements of type `Integer`):

```
import java.util.ArrayList;  
  
public class MyClass {
```

```
public static void main(String[] args) {  
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();  
    myNumbers.add(10);  
    myNumbers.add(15);  
    myNumbers.add(20);  
    myNumbers.add(25);  
    for (int i : myNumbers) {  
        System.out.println(i);  
    }  
}
```

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:

Example

Sort an ArrayList of Strings:

```
import java.util.ArrayList;  
import java.util.Collections; // Import the Collections class  
  
public class MyClass {  
    public static void main(String[] args) {  
        ArrayList<String> cars = new ArrayList<String>();  
        cars.add("Volvo");
```

```
cars.add("BMW");
cars.add("Ford");
cars.add("Mazda");
Collections.sort(cars); // Sort cars
for (String i : cars) {
    System.out.println(i);
}
```

Output:

```
BMW
Ford
Mazda
Volvo
```

Example

Sort an ArrayList of Integers:

```
import java.util.ArrayList;
import java.util.Collections; // Import the Collections class

public class MyClass {
    public static void main(String[] args) {
        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); // Sort myNumbers
```

```
for (int i : myNumbers) {
    System.out.println(i);
}
```

Initialize ArrayList in one line

1.1. Arrays.asList() – Initialize arraylist from array

To initialize an arraylist in single line statement, get all elements in form of **array** using **Arrays.asList** method and pass the array argument to **ArrayList** constructor.

Create arraylist in single statement

```
ArrayList<String> names = new ArrayList<String>( Arrays.asList("alex", "brian", "charles") );
```

```
System.out.println(names);
```

Program output.

```
[alex, brian, charles]
```

1.2. List.of() – Immutable list – Java 9

We can use `List.of()` static factory methods to create immutable lists. Only drawback is that add operation is not supported in these lists.

Create list in single statement

```
List<String> names = List.of("alex", "brian");  
System.out.println(names);
```

Program output.

```
[alex, brian]
```

Read More : [Java 9 Immutable Collections](#)

2. Create ArrayList and add objects – ArrayList constructor

Using **ArrayList constructor** is traditional approach. We create a blank arraylist using constructor and add elements to list using `add()` method. We can add elements either one by one, or we can pass another collection to `add all elements` in one step.

Method	Description
<code>void add(int index, E element)</code>	It is used to insert the specified element at the specified position in a list.
<code>boolean add(E e)</code>	It is used to append the specified element at the end of a list.
<code>boolean addAll(Collection<? extends E> c)</code>	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
<code>boolean addAll(int index, Collection<? extends E> c)</code>	It is used to append all the elements in the specified collection, starting at the specified position of the list.
<code>void clear()</code>	It is used to remove all of the elements from this list.
<code>void ensureCapacity(int requiredCapacity)</code>	It is used to enhance the capacity of an ArrayList instance.

E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.
Object clone()	It is used to return a shallow copy of an ArrayList.
boolean contains(Object o)	It returns true if the list contains the specified element
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
boolean removeIf(Predicate<? super E> filter)	It is used to remove all the elements from the list that satisfies the given predicate.
protected void removeRange(int fromIndex, int toIndex)	It is used to remove all the elements lies within the given range.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.

void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.
Spliterator<E> spliterator()	It is used to create spliterator over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.
int size()	It is used to return the number of elements present in the list.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

Example for removeIf()

```

import java.util.*;
public class GFG {

    public static void main(String[] args)
    {

        // create an ArrayList which going to
        // contains a list of Numbers
        ArrayList<Integer> Numbers = new ArrayList<Integer>();

        // Add Number to list
        Numbers.add(23);
        Numbers.add(32);
        Numbers.add(45);
        Numbers.add(63);

        // apply removeIf() method
        // we are going to remove numbers divisible by 3
        Numbers.removeIf(n -> (n % 3 == 0));
    }
}

```

```
// print list
for (int i : Numbers) {
    System.out.println(i);
}
```

Output:

```
23
```

```
32
```

ArrayList replaceAll() example

Java program to use `replaceAll()` method to transform all the elements of an arraylist using a lambda expression.

2.1. Inline lambda expression

We can use the inline lambda expressions in case we have to execute only single statement.

Convert all strings to lowercase

```
import java.util.ArrayList;
import java.util.Arrays;

public class ArrayListExample
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        ArrayList<String> alphabets = new ArrayList<>(Arrays.asList("A", "B", "C", "D", "E"));

        System.out.println(alphabets);

        alphabets.replaceAll( e -> e.toLowerCase() );
```

```
        System.out.println(alphabets);
    }
}
```

Program output.

```
[A, B, C, D, E]
```

```
[a, b, c, d, e]
```

SubList() Example:

```
import java.util.*;

public class GFG1 {
    public static void main(String[] argv)
        throws Exception
    {
        try {

            // Creating object of ArrayList<Integer>
            ArrayList<String>
                arrlist = new ArrayList<String>();

            // Populating arrlist1
            arrlist.add("A");
            arrlist.add("B");
            arrlist.add("C");
            arrlist.add("D");
            arrlist.add("E");

            // print arrlist
            System.out.println("Original arrlist: "
                + arrlist);

            // getting the subList
            // using subList() method
            List<String> arrlist2 = arrlist.subList(2, 4);

            // print the subList
        }
    }
}
```

```
        System.out.println("Sublist of arrlist: "
                           + arrlist2);
    }

    catch (IndexOutOfBoundsException e) {
        System.out.println("Exception thrown : " + e);
    }

    catch (IllegalArgumentException e) {
        System.out.println("Exception thrown : " + e);
    }
}
```

Output:

Original arrlist: [A, B, C, D, E]

Sublist of arrlist: [C, D]

CONSTRUCTORS IN JAVA

- >Constructor is a special type of method used to initialize the object.
- >Constructor is called, when instance of the class is created.
- >At the time of calling constructor, memory is allocated for the object.
- >Every time an object is created using the new() keyword, at least one constructor is called.
- >Constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).
- >It is called constructor, as it is used to construct the values for an object.

Rules for creating constructor

- >Constructor name should be same as class name and it does not have a return type.
 - >It cannot be static,final,abstract n synchronized.
 - >Access modifiers applicable for constructors are public,default,private,protected.
 - >Constructor's access modifier should be same class's access modifier(for public, default only)
-
- >When we create variable of a class type,only reference is created.
 - >Memory is allocated only when using new() keyword for object creation.

Ex:

```
class Test
{
    // class contents
    void show()
    {
        System.out.println("Test::show() called");
    }
}
```

```
public class Main
{
```

```
// Driver Code  
public static void main(String[] args)  
{  
    Test t;  
  
    // Error here because t  
    // is not initialized  
    t.show();  
}  
}
```

Ex:

```
public class Hello {  
    String name;  
    //Constructor  
    Hello(){  
        this.name = "Hello constructor";  
    }  
    public static void main(String[] args) {  
        Hello obj = new Hello();  
        System.out.println(obj.name);          //prints Hello constructor  
    }  
}
```

Three types of constructors

DEFAULT CONSTRUCTOR

- >When no constructor specified in class, compiler uses the default constructor
- >It will be present in .class file bcz it is inserted in code after compilation only.
- >Body will not be presented explicitly.

NO-ARG CONSTRUCTOR

- >Constructor with no args is called No-arg constructor.
- >Signature is same as default constructor but body for this constructor explicitly exists in program.

Ex:

```
class Demo
{
    public Demo()
    {
        System.out.println("This is a no argument constructor");
    }
    public static void main(String args[])
    {
```

```
    new Demo(); // Prints ,This is a no argument constructor  
}  
}
```

PARAMETERIZED CONSTRUCTOR

>Constructor with args is called PARAMETERIZED constructor.

Ex:

```
public class Employee {  
  
    int empld;  
    String empName;  
  
    //parameterized constructor with two parameters  
    Employee(int id, String name){  
        this.empld = id;  
        this.empName = name;  
    }  
    void info(){  
        System.out.println("Id: "+empld+" Name: "+empName);  
    }  
  
    public static void main(String args[]){  
        Employee obj1 = new Employee(1,"Chaithu");  
    }  
}
```

```
Employee obj2 = new Employee(2,"Nani");

obj1.info();
obj2.info();

}

}
```

O/P:

id:1,name:Chaithu

Id: 1 Name: Chaithu

Id: 2 Name: Nani

Constructor Overloading

>It applies when one or more constructors present in a class with different parameters.

>Constructor calling must be the first statement of constructor in Java.

>If we have defined any parameterized constructor, then compiler will not create default constructor.

and vice versa if we don't define any constructor, the compiler creates the default constructor by default during compilation

```
// Java program to illustrate
// Constructor Overloading
class Box
{
    double width, height, 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 = height = depth = 0;  
}
```

```
// constructor used when cube is created  
Box(double len)  
{  
    width = height = depth = len;
```

```
// compute and return volume  
double volume()  
{  
    return width * height * depth;
```

```
    }

}

public class Test
{
    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);
    }
}
```

```
 }  
}
```

Constructor Chaining:

- >The this() expression should always be the first line of the constructor.
- >There should be at-least be one constructor without the this() keyword (constructor 3 in above example).
- >Constructor chaining can be achieved in any order.
- >It is used when we want to perform multiple tasks in a single constructor

Constructor Chaining with this()

- >Constructor chaining is the process of calling one constructor from another constructor with respect to current object.
- >Constructor chaining can be done in two ways:
 - Within same class: It can be done using this() keyword for constructors in same class
 - From base class: by using super() keyword to call constructor from the base class.

Ex:

```
----  
  
// Java program to illustrate Constructor Chaining  
// within same class Using this() keyword
```

```
class Temp
{
    // default constructor 1
    // default constructor will call another constructor
    // using this keyword from same class
    Temp()
    {

        System.out.println("The Default constructor");
    }

    // parameterized constructor 2
    Temp(int x)
    {

        this();      /// invokes default constructor
        System.out.println(x);
    }

    // parameterized constructor 3
    Temp(int x, int y)
    {
        this(5);    // invokes parameterized constructor 2
        System.out.println(x * y);
    }

    public static void main(String args[])
}
```

```
{  
    // invokes parameterized constructor 3 first  
    new Temp(8,10);  
}  
}
```

Constructor Chaining using super()

- >super() shpuld be in the first line in a constructor
- >Class should follow inheritance to implement super()

Ex:

```
----  
  
// Java program to illustrate Constructor Chaining to  
// other class using super() keyword  
  
class Base  
{  
    String name;  
  
    // constructor 1  
    Base()  
    {  
        this("");  
        System.out.println("No-argument constructor of base class");  
    }  
}
```

```
// constructor 2
Base(String name)
{
    this.name = name;
    System.out.println("Calling parameterized constructor of base");
}

class Derived extends Base
{
    // constructor 3
    Derived()
    {
        System.out.println("No-argument constructor of derived");
    }

    // parameterized constructor 4
    Derived(String name)
    {
        // invokes base class constructor 2
        super(name);
        System.out.println("Calling parameterized constructor of derived");
    }

    public static void main(String args[])
    {
        // calls parameterized constructor 4
    }
}
```

```
Derived obj = new Derived("test");

// Calls No-argument constructor
// Derived obj = new Derived();

}

}
```

CONSTRUCTOR OVERRIDING:

>It is not applicable bcz,If we want to implement constructor overriding,Parent class's constructor has to be defined in the child class which is not applicable due to constuctor rules.

Ex:

```
-----
public class Parent
{
    Parent()
    {
        System.out.println("Parent class constructor");
    }
}

class Child extends Parent
{
    Parent()
    {
        System.out.println("Child class constructor"); //throws an
error bcz constructor name should be same as class's name.
```

```
 }  
}
```

DATA TYPES

data type defines type of variable and memory size of a variable and range value

- 2) to represent numeric values we have byte(1 bytes),short(2 bytes),int(4) long(8)
- 3) to represent floating values we have float*(4) and double(8)
- 4) to define single character we have char(2)

5) to represent true or false we have boolean(no size,jvm will decide)

these are the 8 primitive data types

we have 8 bits for one byte 0-6 bits will represent value and last 7th bit will represent sign (0 for +ve and 1 for -ve

to find range we have formula -2^n to $((2^n)-1)$

```
int a=5 //4 bytes address(100)->5
```

```
short b=5// 2bytes
```

```
byte c=5 //1 byte
```

```
float d=5.5f //4bytes
```

```
double e=5.5 //8bytes
```

```
or double f=5
```

```
long g=1234l//8 bytes
```

```
char a='E'
```

```
a=65;
```

```
System.out.println(a);//A
```

```
double a=5;
```

```
System.out.println(a)//5.0(implicit)
```

```
int a=(int)5.6;  
System.out.println(a)//5(explicit)
```

byte-->short-->int-->long-->float-->double

char-->

default values for datatypes:

int a; 0

char c; single space

double d; 0.0

boolean d; false

char d;\u0000

declaring double to float is compilation error

float a= (float)10.5

Non Primitive DataTypes:

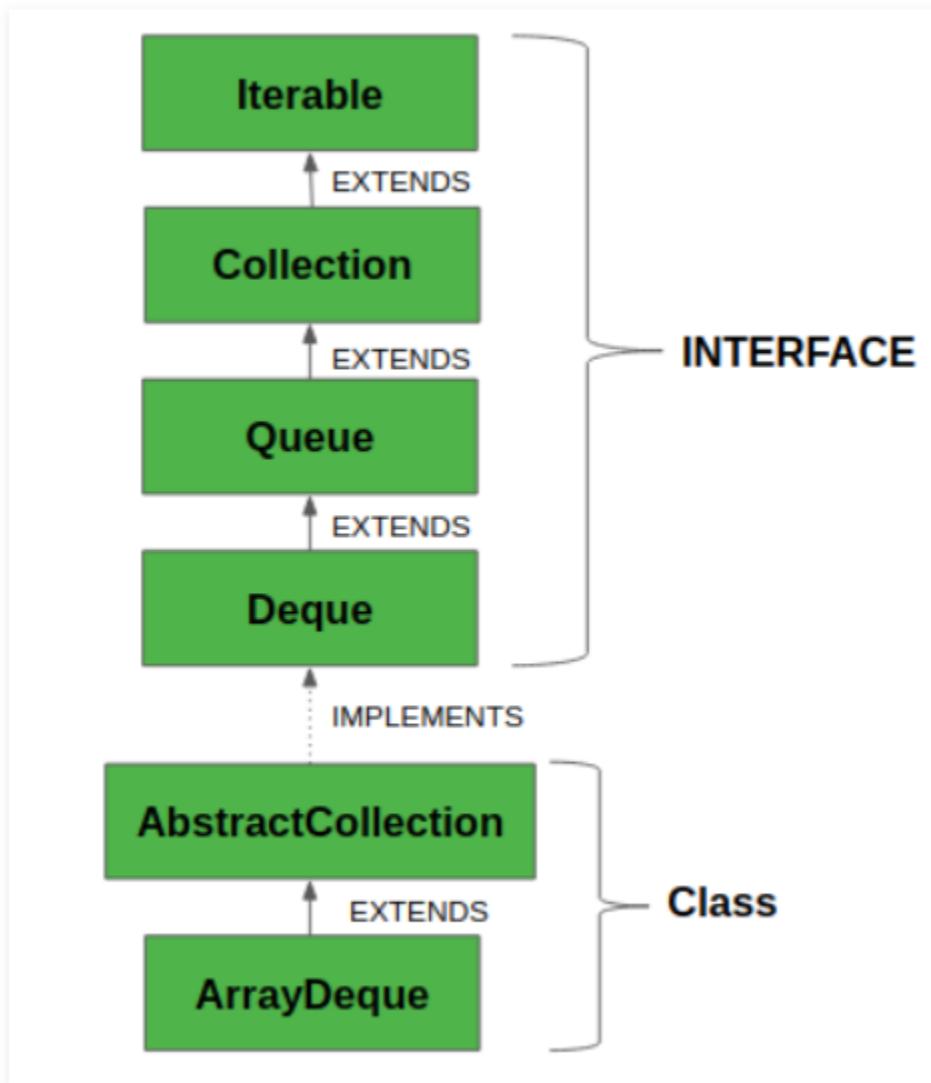
non primitive datatypes stores reference ,it may be class ,interface, array variable
it refers memory location where data is stored

also called reference variable or object reference

String, array and class are non primitive types

DEQUE INTERFACE IN JAVA

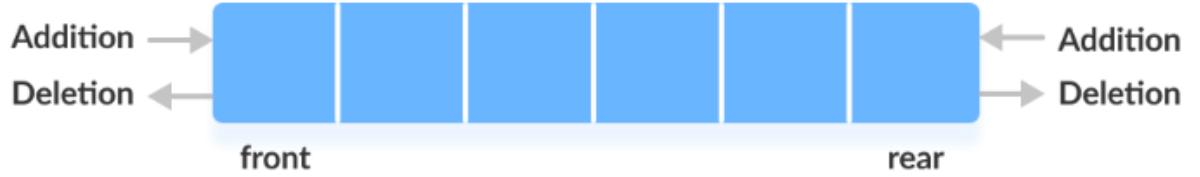
- The Deque interface of the Java collections framework provides the functionality of a double-ended queue. It extends the Queue interface.
- Hierarchy of Deque interface



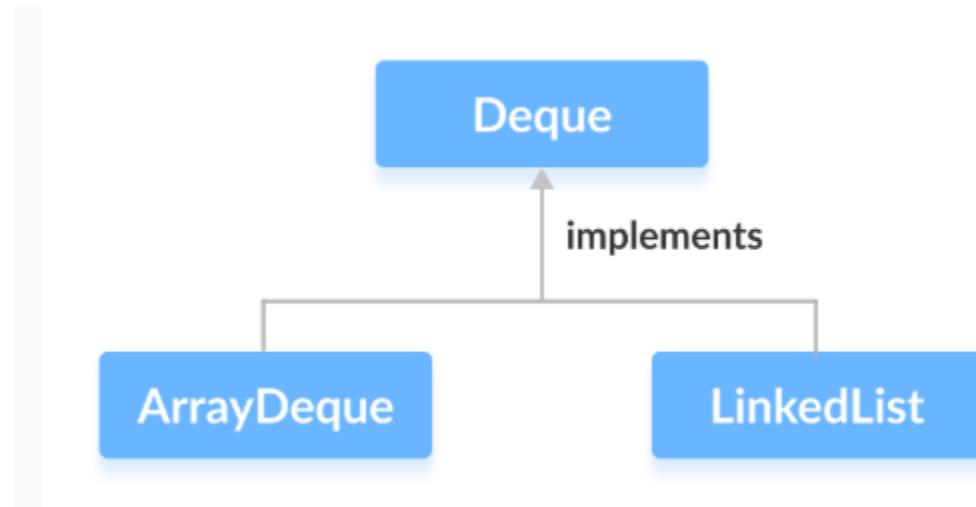
Few important features of Deque are:

- It provides the support of resizable array and helps in restriction-free capacity, so to grow the array according to the usage.
- Array deques prohibit the use of Null elements and do not accept any such elements.
- Any concurrent access by multiple threads is not supported.
- In the absence of external synchronization, Deque is not thread-safe.

- Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for "**double ended queue**".



- In order to use the functionalities of the Deque interface, we need to use classes that implement it:



- The `LinkedList` class is a pretty standard Deque and Queue implementation. It uses a linked list internally to model a queue or a deque.
- The Java `ArrayDeque` class stores its elements internally in an array. If the number of elements exceeds the space in the array, a new array is allocated, and all elements moved over. In other words, the `ArrayDeque` grows as needed, even if it stores its elements in an array.
- In Java, we must import the `java.util.Deque` package to use `Deque`.

```
// Array implementation of Deque
Deque<String> animal1 = new ArrayDeque<>();
```

```
// LinkedList implementation of Deque  
Deque<String> animal2 = new LinkedList<>();
```

- By default you can put any Object into a Java Deque. However, using Java Generics it is possible to limit the types of object you can insert into a Deque. Here is an example:

```
Deque<MyObject> deque = new LinkedList<MyObject>();
```

This Deque can now only have MyObject instances inserted into it. You can then access and iterate its elements without casting them. Here is how it looks:

```
MyObject myObject = deque.remove();  
  
for(MyObject anObject : deque){  
    //do something to anObject...  
}
```

Methods of Dequeue:

add(element): Adds an element to the tail.

addFirst(element): Adds an element to the head.

addLast(element): Adds an element to the tail.

offer(element): Adds an element to the tail and returns a boolean to explain if the insertion was successful.

offerFirst(element): Adds an element to the head and returns a boolean to explain if the insertion was successful.

offerLast(element): Adds an element to the tail and returns a boolean to explain if the insertion was successful.

iterator(): Returns an iterator for this deque.

descendingIterator(): Returns an iterator that has the reverse order for this deque.

push(element): Adds an element to the head.

pop(element): Removes an element from the head and returns it.

removeFirst(): Removes the element at the head.

removeLast(): Removes the element at the tail.

poll(): Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty.

pollFirst(): Retrieves and removes the first element of this deque, or returns null if this deque is empty.

pollLast(): Retrieves and removes the last element of this deque, or returns null if this deque is empty.

peek(): Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty.

peekFirst(): Retrieves, but does not remove, the first element of this deque, or returns null if this deque is empty.

peekLast(): Retrieves, but does not remove, the last element of this deque, or returns null if this deque is empty.

Summarizing the methods, we get:

Operations	First Element or Head		Last Element or Tail	
	Throws exception	Special Value	Throws exception	Special Value
Insert	addFirst(element)	offerFirst(element)	addLast(element)	offerLast(element)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

Example:

```
// Java program to demonstrate working of
// Deque in Java
import java.util.*;
public class Main
{
    public static void main(String[] args)
    {
        Deque<String> deque = new ArrayDeque<String>();
```

```
// We can add elements to the queue in various ways
deque.add("Element 1 (Tail)"); // add to tail
deque.addFirst("Element 2 (Head)");
deque.addLast("Element 3 (Tail)");
deque.push("Element 4 (Head)"); //add to head
deque.offer("Element 5 (Tail)");
deque.offerFirst("Element 6 (Head)");
deque.offerLast("Element 7 (Tail)");

// System.out.println("Deque " + "\n");

// Iterate through the queue elements.
System.out.println("Standard Iterator");
Iterator iterator = deque.iterator();
while (iterator.hasNext())
    System.out.println("\t" + iterator.next());

// Reverse order iterator
Iterator reverse = deque.descendingIterator();
System.out.println("Reverse Iterator");
while (reverse.hasNext())
    System.out.println("\t" + reverse.next());

// Peek returns the head, without deleting
// it from the deque
System.out.println("Peek " + deque.peek());
System.out.println("Peek " + deque.peekFirst());
System.out.println("Peek " + deque.peekLast());
```

```
System.out.println("After peek: " + deque);

// Pop returns the head, and removes it from
// the deque
System.out.println("Pop " + deque.pop());
System.out.println("After pop: " + deque);

// We can check if a specific element exists
// in the deque
System.out.println("Contains element 3: " +
deque.contains("Element 3 (Tail)"));

// We can remove the first / last element.
System.out.println(deque.removeFirst());
System.out.println(deque.removeLast());
System.out.println("Deque after removing " +
"first and last: " + deque);

//poll methods
System.out.println(deque.poll());
System.out.println(deque.pollFirst());
System.out.println(deque.pollLast());
System.out.println("deque after using poll methods: " + deque);

}

}
```

Output:

```
Standard Iterator
Element 6 (Head)
Element 4 (Head)
Element 2 (Head)
Element 1 (Tail)
Element 3 (Tail)
Element 5 (Tail)
Element 7 (Tail)
Reverse Iterator
Element 7 (Tail)
Element 5 (Tail)
Element 3 (Tail)
Element 1 (Tail)
Element 2 (Head)
Element 4 (Head)
Element 6 (Head)
Peek Element 6 (Head)
Peek Element 6 (Head)
Peek Element 7 (Tail)
After peek: [Element 6 (Head), Element 4 (Head), Element 2 (Head), Element 1 (Tail), Element 3 (Tail), Element 5 (Tail), Element 7 (Tail)]
Pop Element 6 (Head)
After pop: [Element 4 (Head), Element 2 (Head), Element 1 (Tail), Element 3 (Tail), Element 5 (Tail), Element 7 (Tail)]
Contains element 3: true
Element 4 (Head)
Element 7 (Tail)
Deque after removing first and last: [Element 1 (Tail), Element 3 (Tail), Element 5 (Tail)]
Element 1 (Tail)
Element 3 (Tail)
Element 5 (Tail)
deque after using poll methods: []
```

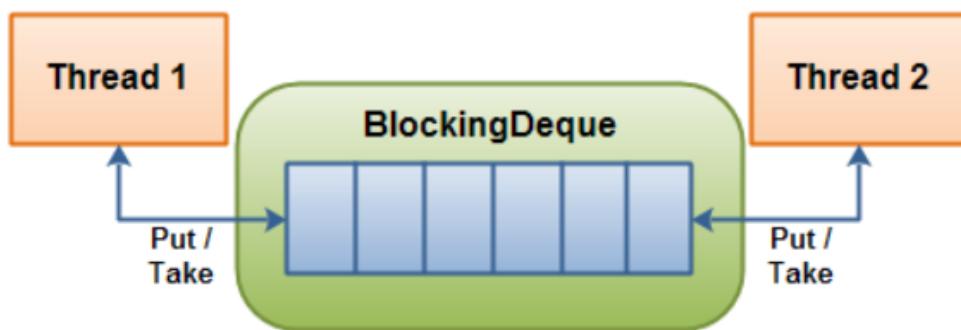
Blocking deque:

- The BlockingDeque interface in the `java.util.concurrent` class represents a deque which is thread safe to put into, and take instances from.
- The BlockingDeque class is a Deque which blocks threads trying to insert or remove elements from the deque, in case it is either not possible to insert or remove elements from the deque.
- Blocking deque interface extends deque interface, inserting and deleting elements in deque will be doubled as we can insert/delete elements at both sides in deque.
- Here is a table of what the methods of the BlockingQueue does in a BlockingDeque implementation:

BlockingQueue	BlockingDeque
add()	addLast()
offer() x 2	offerLast() x 2
put()	putLast()
remove()	removeFirst()
poll() x 2	pollFirst()
take()	takeFirst()
element()	getFirst()
peek()	peekFirst()

Blocking deque usage:

A BlockingDeque could be used if threads are both producing and consuming elements of the same queue. It could also just be used if the producing thread needs to insert at both ends of the queue, and the consuming thread needs to remove from both ends of the queue. Here is an illustration of that:



A BlockingDeque - threads can put and take from both ends of the deque.

Explanation:

A thread will produce elements and insert them into either end of the queue. If the deque is currently full, the inserting thread will be blocked until a removing thread takes

an element out of the deque. If the deque is currently empty, a removing thread will be blocked until an inserting thread inserts an element into the deque.

Blocking deque methods:

- A BlockingDeque has 4 different sets of methods for inserting, removing and examining the elements in the deque.

	Throws Exception	Special Value	Blocks	Times Out
Insert	<code>addFirst(o)</code>	<code>offerFirst(o)</code>	<code>putFirst(o)</code>	<code>offerFirst(o, timeout, timeunit)</code>
Remove	<code>removeFirst(o)</code>	<code>pollFirst(o)</code>	<code>takeFirst(o)</code>	<code>pollFirst(timeout, timeunit)</code>
Examine	<code>getFirst(o)</code>	<code>peekFirst(o)</code>		
	Throws Exception	Special Value	Blocks	Times Out
Insert	<code>addLast(o)</code>	<code>offerLast(o)</code>	<code>putLast(o)</code>	<code>offerLast(o, timeout, timeunit)</code>
Remove	<code>removeLast(o)</code>	<code>pollLast(o)</code>	<code>takeLast(o)</code>	<code>pollLast(timeout, timeunit)</code>
Examine	<code>getLast(o)</code>	<code>peekLast(o)</code>		

- Throws Exception:**

If the attempted operation is not possible immediately, an exception is thrown.

- Special Value:**

If the attempted operation is not possible immediately, a special value is returned (often true / false).

- Blocks:**

If the attempted operation is not possible immediately, the method call blocks until it is.

- Times Out:**

If the attempted operation is not possible immediately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

BlockingDeque Code Example

- Here is a small code example of how to use the BlockingDeque methods:

```
// Java Program Demonstrate some of Blocking deque methods
```

```
// method of BlockingDeque

import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.BlockingDeque;
import java.util.*;

public class Main {
    public static void main(String[] args)
        throws InterruptedException
    {

        // create object of BlockingDeque
        BlockingDeque<Integer> bd
            = new LinkedBlockingDeque<Integer>(6);

        // Add numbers to BlockingDeque
        bd.putFirst(2);
        bd.putLast(3);
        bd.addFirst(5);
        bd.addLast(4);
        bd.offerFirst(6);
        bd.offerLast(7);

        try
        {
            bd.offerLast(10);
        } catch(Exception e)
        {
            System.out.println(e);
        }

        // print Deque
        System.out.println("Blocking Deque: " +bd);

        //remove elements
        System.out.println( bd.removeFirst());
        System.out.println(bd.takeFirst());
        System.out.println(bd.pollFirst());

        // print Deque
        System.out.println("Blocking Deque: " +bd);
```

```

        System.out.println( bd.removeLast());
        System.out.println(bd.takeLast());
        System.out.println(bd.pollLast());

        //get elements
        System.out.println(bd.peekFirst());
        System.out.println(bd.getFirst());

        // print Deque
        System.out.println("Blocking Deque: " +bd);

    }

}

```

Output:

```

Blocking Deque: [6, 5, 2, 3, 4, 7]
6
5
2
Blocking Deque: [3, 4, 7]
7
4
3
null
Exception in thread "main" java.util.NoSuchElementException
    at java.util.concurrent.LinkedBlockingDeque.getFirst (LinkedBlockingDeque.java:553)
    at Main.main(Main.java:51)

```

Since, Blocking deque is an interface, we need one of its implementations to use it. The `java.util.concurrent` package has **LinkedBlockingDeque** which will implement `BlockingDeque` interface

[Linked Blocking Deque:](#)

- The `LinkedBlockingDeque` class implements the `BlockingDeque` interface.
- The `LinkedBlockingDeque` is a `Deque` which will block if a thread attempts to take elements out of it while it is empty, regardless of what end the thread is attempting to take elements from.
- Here, if we insert null values, throws `nullpointerexception`.

Constructors in Java LinkedBlockingDeque:

- **LinkedBlockingDeque():** This constructor is used to construct an empty deque. In this case the capacity is set to Integer.MAX_VALUE
- **LinkedBlockingDeque(int capacity):** This constructor creates a LinkedBlockingDeque with the given (fixed) capacity.
- **LinkedBlockingDeque(Collection<E> c):** This constructor is used to construct a deque with the elements of the Collection passed as the parameter.

Example:

```
import java.util.concurrent.LinkedBlockingDeque;  
import java.util.*;  
  
public class Main {  
    public static void main(String[] args)  
        throws InterruptedException  
    {  
  
        // create object of LinkedBlockingDeque  
        // using LinkedBlockingDeque() constructor  
        LinkedBlockingDeque<Integer> LBD  
            = new LinkedBlockingDeque<Integer>();  
  
        // Add numbers to end of LinkedBlockingDeque  
        LBD.addFirst(7855642);  
        LBD.add(35658786);  
        LBD.add(5278367);  
        LBD.addLast(74381793);  
  
        // print Dequee  
        System.out.println("Linked Blocking Deque1: "
```

```
        + LBD);  
System.out.println("Size of Linked Blocking Deque1: "  
        + LBD.size());  
  
// create object of LinkedBlockingDeque  
// using LinkedBlockingDeque(int capacity) constructor  
LinkedBlockingDeque<Integer> LBD1  
= new LinkedBlockingDeque<Integer>(3);  
  
// Add numbers to end of LinkedBlockingDeque  
LBD1.add(7855642);  
LBD1.add(35658786);  
LBD1.add(5278367);  
  
try {  
    // adding the 4th element  
    // will throw exception for Deque full  
    LBD1.add(74381793);  
}  
catch (Exception e) {  
    System.out.println("Exception: " + e);  
}  
try {  
    // adding the 4th element  
    // will throw exception for Deque full  
    LBD1.add(null);  
}  
catch (Exception e) {
```

```
        System.out.println("Exception: " + e);
    }

    // print Dequee
    System.out.println("Linked Blocking Deque2: "
        + LBD1);

    System.out.println("Size of Linked Blocking Deque2: "
        + LBD1.size());

    // take and put methods
    //since, LBD2 size is 1, it cant insert more than one elements
    LinkedBlockingDeque<Integer> LBD2
        = new LinkedBlockingDeque<Integer>(1);

    LBD2.add(123);

    // print Dequee
    System.out.println("Linked Blocking Deque3: "
        + LBD2);

    System.out.println(LBD2.takeFirst());

    //if we enter one more takeFirst stmt, it will wait upto sometime for an element
    //System.out.println(LBD2.takeFirst());

    LBD.putFirst(5278367);
    System.out.println("Linked Blocking Deque3: "
        + LBD2);
```

```
}
```

```
}
```

Output:

```
Linked Blocking Deque1: [7855642, 35658786, 5278367, 74381793]
Size of Linked Blocking Deque1: 4
Exception: java.lang.IllegalStateException: Deque full
Exception: java.lang.NullPointerException
Linked Blocking Deque2: [7855642, 35658786, 5278367]
Size of Linked Blocking Deque2: 3
Linked Blocking Deque3: [123]
123
Linked Blocking Deque3: []
```

DIFFERENCES BETWEEN CLASS ABSTRACT CLASS AND INTERFACE

- >If we dont know anything abt implementation, we need to choose Interface.
- >If we are talking abt implementation ,but not completely, then we need to choose abstract class.
- >If we know the implementation and methods are ready to provide response, then we need to choose concrete class.

ABSTRACT CLASS	CONCRETE CLASS
An abstract class is declared using abstract modifier.	Concrete class is created using class keyword only.
An abstract class cannot be directly instantiated using the new keyword.	A concrete class can be directly instantiated using the new keyword.
Abstract class can have both an abstract as well as concrete methods	A concrete class can only have concrete methods. Even a single abstract method makes the class abstract.
An abstract class cannot be final, because all its abstract methods must defined in the subclass.	A concrete class can be declared as final.

Abstract class can not implement an interface alone.

A child class is needed to be able to use the interface for instantiation.

Interface can be implemented easily.

Ex:

```
abstract class DemoAbstractClass {
```

```
    abstract void display();
```

```
}
```

```
public class Main {
```

```
    public static void main(String[] args)
```

```
{
```

```
    DemoAbstractClass AC = new DemoAbstractClass(){
```

```
        void display()
```

```
{
```

```
    System.out.println("Hii");
```

```
}
```

```
};
```

```
System.out.println("Hello");
```

```
AC.display();
```

```
}
```

```
}
```

O/p:

Hello

Hii

DIFFERENCES BETWEEN Interface AND ABSTRACT CLASS

Interface	ABSTRACT CLASS
If we dont know anything abt implementation i.e just know requirements, we need to choose Interface.	If we are talking abt implementation ,but not completely, then we need to choose abstract class.
Inside interface, every method should always be public and abstract.	Every methods present in abstract class need not to be public and abstract.
Interface methods cannot declare with private,protected access modifiers.	There are no restrictions on abstract class method access modifiers.
Every interace variable is public,static and final.	Abstract class can have final, non-final, static and non-static variables.
We cant declare interface variables with modifiers like private,protected,transient and volatile.	There are no restrictions on Abstract class variable modifiers
For the interface variables,we should perform intialization at the time of declaration otherwise we will get CE.	For abstract class variables,it is not required to perform intialization.
Inside the interface we cant declare instance and static blocks if we declare,it throws CE.	Inside abstract class, we can declare instance and static blocks.

Constructors are not valid inside interface as the variables are already initialised during declaration itself.	Inside abstract class, we can declare constructor which handles object creation.
It contains methods only without any definition.	It can contain concrete methods in addition to abstract methods.

As we cannot create object for interfaces, serialization concept is invalid for interfaces.

Ex:

```
interface Animal
```

```
{
```

```
    public void eat();
```

```
    public static final int a=9;
```

```
}
```

```
abstract class Hello implements Animal
```

```
{
```

```
    public void eat()
```

```
{
```

```
    System.out.println("eating");
```

```
}
```

```
    abstract void display();
```

```
}
```

```
public class Main
```

```
{
```

```

public static void main(String[] args) {
    System.out.println("Hello World");
    Hello h=new Hello()
    {
        void display()
        {
            System.out.println("displaying");
        }
    };
    h.eat();
    h.display();
    System.out.println(Animal.a);
}

```

O/P:

Hello World

eating

displaying

9

DIFFERENCES BETWEEN CLASS AND INTERFACE

CLASS	Interface
A class can be instantiated i.e, objects of a class can be created.	An Interface cannot be instantiated i.e, objects cannot be created.

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

Ex:

```
// Java program to demonstrate
// working of interface.
```

```
import java.io.*;
```

```
// A simple interface
interface in1
{
```

```
// public, static and final
```

```
final int a = 10;

// public and abstract
void display();

}

// A class that implements the interface.
class testClass implements in1

{

    // Implementing the capabilities of
    // interface.

    public void display()

    {

        System.out.println("Geek");

    }

}

// Driver Code
public static void main(String[] args)
{
    testClass t = new testClass();
    t.display();
    System.out.println(a);
}
}
```

O/P:

Geek

REFERENCE LINKS:

<https://www.geeksforgeeks.org/differences-between-interface-and-class-in-java/>

<https://www.tutorialspoint.com/differences-between-abstract-class-and-concrete-class-in-java>

<https://www.geeksforgeeks.org/difference-between-abstract-class-and-interface-in-java/>

EXCEPTION HANDLING IN JAVA

Exception

- An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. In such cases we get a system generated error message.
- The good thing about exceptions is, by handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message to the user.

Why an exception occurs?

- There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

Advantage of exception handling

- Exception handling ensures that the flow of the program doesn't break when an exception occurs.
- For example, if a program has bunch of statements and an exception occurs mid way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly.
- By handling we make sure that all the statements execute and the flow of program doesn't break.

Difference between error and exception

Errors indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.

Exceptions are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions. Few examples:

NullPointerException – When you try to use a reference that points to null.

ArithmaticException – When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.

ArrayIndexOutOfBoundsException – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.

Types of Exceptions

There are 2 types of exceptions namely checked and unchecked.

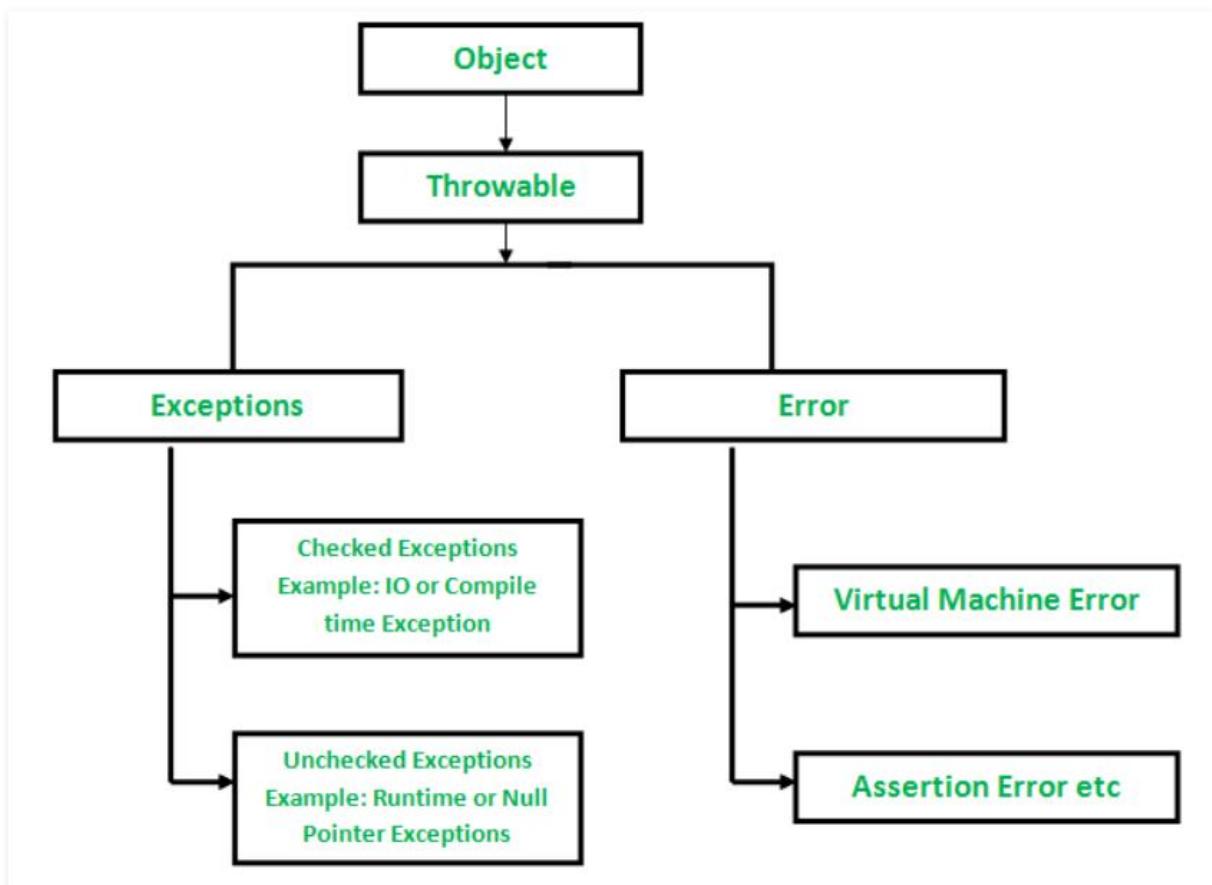
Checked exceptions

- All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error. For example, SQLException, IOException, ClassNotFoundException etc.

Unchecked Exceptions

- Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it's the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmaticException, NullPointerException, ArrayIndexOutOfBoundsException

etc.

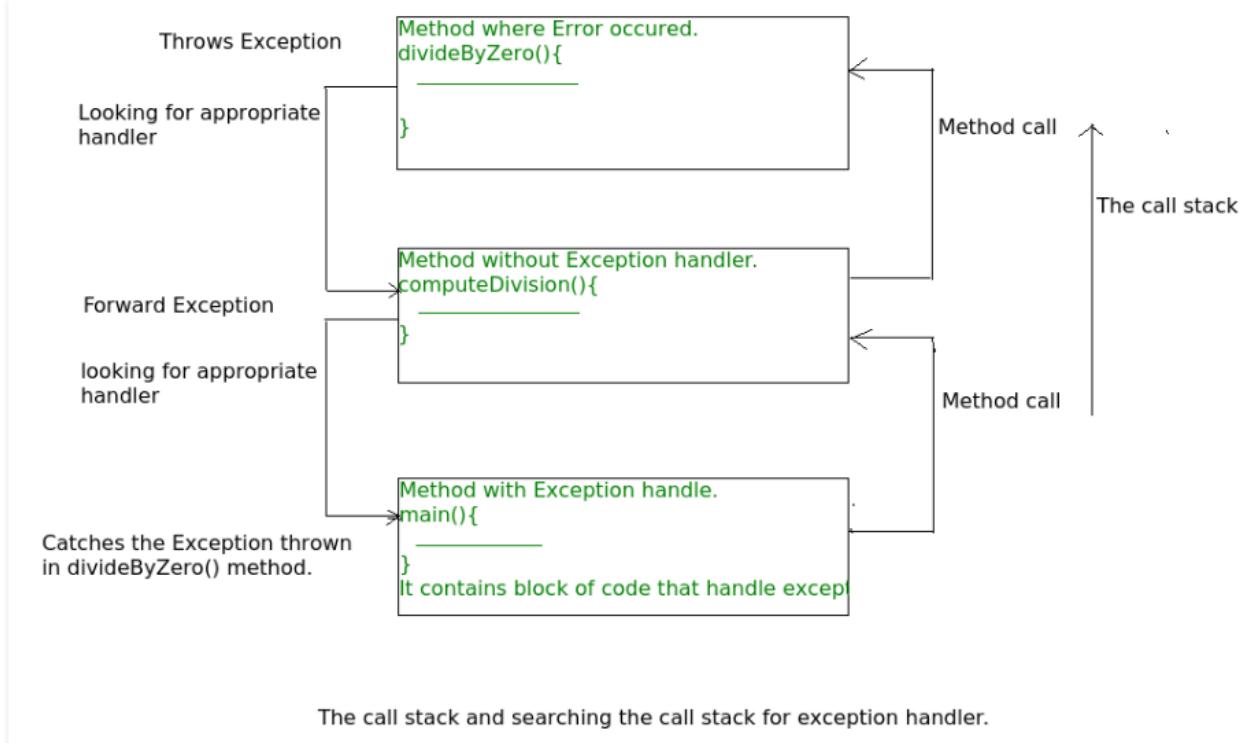


How JVM handle an Exception?

Default Exception Handling :

- Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system(JVM). The exception object contains name and description of the exception, and current state of the program where exception has occurred. Creating the Exception Object and handing it to the run-time system is called throwing an Exception. There might be the list of the methods that had been called to get to the method where exception was occurred. This ordered list of the methods is called **Call Stack**.

See the below diagram to understand the flow of the call stack.



How Programmer handles exception

Customized Exception Handling :

- Java exception handling is managed via five keywords: try, catch, [throw](#), [throws](#), and finally. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword [throw](#). Any exception that is thrown out of a method must be specified as such by a [throws](#) clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

Try Block

- The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

Syntax of try block

```
try{
    //statements that may cause an exception
}
```

Catch Block

- A catch block is where you handle the exceptions, this block must follow the try block. A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

Syntax of try-catch block in java

```
try
{
    //statements that may cause an exception
}
catch (exception(type) e(object))
{
    //error handling code
}
```

Example of try-catch block:

```
public class Main

{
    public static void main(String[] args) {
        System.out.println("Hello World");

        try
        {

```

```

int a[]={};
a[4]=30/0;
System.out.println("First print statement in try block");

}

catch(ArithmetricException e)
{
    e.printStackTrace();
    System.out.println("Warning: ArithmetricException");
}

}

}

Output:
```

```
java.lang.ArithmetricException: / by zero
        at Main.main(Main.java:16)
Warning: ArithmetricException
```

Multiple catch blocks:

- When a program is handling multiple exceptions, we use multiple catch blocks.
- However, Generic catch block can handle all kinds of exceptions, but if we want to specify a particular exception occurred we need to use multiple catch blocks.
- Generic catch block should be the last block among all the catch blocks otherwise it throws a compile time error.

Example:

```
public class Main

{
    public static void main(String[] args) {
        System.out.println("Hello World");

        try
        {
            int a[]=new int[7];
            a[4]=30/0;

            System.out.println("First print statement in try block");
        }

        catch(ArithmaticException e)
        {
            e.printStackTrace();
            System.out.println("Warning: ArithmaticException");
        }

        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Warning: ArrayIndexOutOfBoundsException");
        }

        catch(Exception e)
```

```

{
    System.out.println("Warning: Some Other exception");
    e.printStackTrace();
}

}
}

```

Output:

```

Hello World
java.lang.ArithmetricException: / by zero
        at Main.main(Main.java:16)
Warning: ArithmetricException

```

Nested try-catch:

- When a **try catch block** is present in another try block then it is called the nested try catch block. Each time a try block does not have a catch handler for a particular **exception**, then the catch blocks of parent try block are inspected for that exception, if match is found that that catch block executes.
- If neither catch block nor parent catch block handles exception then the system generated message would be shown for the exception
- Often **nested try** statements are used to allow different categories of errors to be handled in different ways. Many programmers **use** an inner **try block** to **catch** the less severe errors and outer **try block** to **catch** the more severe errors.

Syntax:

```
//Main try block
try {
    statement 1;
    statement 2;
    //try-catch block inside another try block
    try {
        statement 3;
        statement 4;
        //try-catch block inside nested try block
        try {
            statement 5;
            statement 6;
        }
        catch(Exception e2) {
            //Exception Message
        }
    }
    catch(Exception e1) {
        //Exception Message
    }
}
//Catch of Main(parent) try block
catch(Exception e3) {
    //Exception Message
}
```

Example:

```
{
    public static void main(String[] args) {
        System.out.println("Hello World");
    }

    try
    {
        //try-block2
        try
```

```
{\n    //try-block3\n    try{\n        int arr[] = {1,2,3,4};\n\n        System.out.println(arr[10]);\n    }\n\n    catch(ArithmeticException e)\n    {\n        System.out.print("Arithmetic Exception");\n        System.out.println(" handled in try-block3");\n    }\n\n    }\n\n    catch(ArrayIndexOutOfBoundsException e4)\n    {\n        System.out.print("ArrayIndexOutOfBoundsException");\n        System.out.println(" handled in try-block2");\n    }\n\n    catch(ArithmeticException e)\n    {\n        System.out.print("Arithmetic Exception");\n        System.out.println(" handled in try-block2");\n    }\n}
```

```
    }

}

catch(ArithmeticException e3)

{

System.out.print("Arithmetic Exception");

System.out.println(" handled in main try-block");

}

catch(ArrayIndexOutOfBoundsException e4)

{

System.out.print("ArrayIndexOutOfBoundsException");

System.out.println(" handled in main try-block");

}

catch(Exception e5)

{

System.out.print("Exception");

System.out.println(" handled in main try-block");

}

}
```

Output:

```
Hello World
ArrayIndexOutOfBoundsException handled in try-block2
```

Throwing an exception

- We can define our own set of conditions or rules and throw an exception explicitly using throw keyword.
- For example, we can throw ArithmeticException when we divide number by 5, or any other numbers, what we need to do is just set the condition and throw any exception using throw keyword.
- We can throw user-defined exceptions also using this throw keyword.

Syntax of throw keyword:

```
throw new exception_class("error message");
```

Example

```
throw new ArithmeticException("dividing a number by 5 is not allowed in this program");
```

Program:

```
public class Main

{
    static void checkEligibility(int stuage, int stuweight)
    {
        if(stuage<12 && stuweight<40)
        {
            throw new ArithmeticException("Student is not eligible for registration");
        }
    }
}
```

```
else
{
    System.out.println("Student Entry is Valid!!");

}
}

public static void main(String[] args)
{
    System.out.println("Exception handling in java");
    int num1, num2;

    System.out.println("Welcome to the Registration process!!");

    checkEligibility(15, 49);
    checkEligibility(10, 39);

    System.out.println("Have a nice day.. ");

}
```

Output:

```
Exception handling in java
Welcome to the Registration process!!
Student Entry is Valid!!
Exception in thread "main" java.lang.ArithmetricException: Student is not eligible for registration
        at Main.checkEligiblty(Main.java:13)
        at Main.main(Main.java:43)
```

Throws Keyword in java:

- The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- throws keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program.

Syntax of throws

```
return_type method_name() throws exception_class_name{
    //method code
}
```

Advantage of java throws keyword:

- Now Checked Exception can be propagated (forwarded in call stack).
- It provides information to the caller of the method about the exception.

Rule: If you are calling a method that declares an exception, you must either catch or declare the exception.

There are two cases:

1. **Case1:** You caught the exception i.e. handle the exception using try/catch.
2. **Case2:** You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```

1. import java.io.*;
2. class M{
3.     void method()throws IOException{
4.         throw new IOException("device error");
5.     }
6. }
7. public class Testthrows2{
8.     public static void main(String args[]){
9.         try{
10.             M m=new M();
11.             m.method();
12.         }catch(Exception e){System.out.println("exception handled");}
13.
14.         System.out.println("normal flow...");}
15.     }
16. }
```

Output:

Exception handled

Normal flow...

Case2: You declare the exception

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- B)In case you declare the exception if exception occurs, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```

1. import java.io.*;
2. class M{
3.     void method()throws IOException{
```

```
4. System.out.println("device operation performed");
5. }
6. }
7. class Testthrows3{
8.   public static void main(String args[])throws IOException{//declare exception
9.     M m=new M();
10.    m.method();
11.
12.   System.out.println("normal flow...");
13. }
14. }
```

Output:

device operation performed
normal flow...

B)Program if exception occurs

```
1. import java.io.*;
2. class M{
3.   void method()throws IOException{
4.     throw new IOException("device error");
5.   }
6. }
7. class Testthrows4{
8.   public static void main(String args[])throws IOException{//declare exception
9.     M m=new M();
10.    m.method();
11.
12.   System.out.println("normal flow...");
13. }
14. }
```

Output:

Runtime Exception

Finally Keyword in java:

- A **finally block** contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing a connection, stream etc.
- A finally block must be associated with a try block, you cannot use finally without a try block. You should place those statements in this block that must be executed always.
- An exception in the finally block, behaves exactly like any other exception.
- The statements present in the **finally block** execute even if the try block contains control transfer statements like return, break or continue.

Syntax of Finally block

```
try {
    //Statements that may cause an exception
}
catch {
    //Handling exception
}
finally {
    //Statements to be executed
}
```

Example:

```
class JavaFinally
{
    public static void main(String args[])
    {
        System.out.println(JavaFinally.myMethod());
    }
    public static int myMethod()
    {
        try {
            return 112;
        }
        finally {
            System.out.println("This is Finally block");
            System.out.println("Finally block ran even after return statement");
        }
    }
}
```

Output of above program:

```
This is Finally block
Finally block ran even after return statement
112
```

Cases when finally block don't execute:

The circumstances that prevent execution of the code in a finally block are:

- The death of a thread.
- Use of System.exit(0) method when exception not raised in try block.
- When exception arised in a finally block

Example:

Using System.exit(0) without arising an exception

```
public class Main

{
    public static void main(String[] args) {
        System.out.println("Hello World");
        int b=0;
        try {
            System.out.println("stmt in try block");
            // int a=40/0;
            System.exit(0);
        }
        catch(ArithmeticException e)
        {
            System.out.println("stmt in catch block");
            System.out.println(e);
        }
    }
}
```

```

        } finally

    {

        System.out.println("stmt in finally block");

    }

}

```

Output:

```
Hello World
stmt in try block
```

- But if any exception raised(40/0), even placing the system.exit(0) cannot stop the execution of finally block.
- Below is the output for un-commenting (40/0).

```
Hello World
stmt in try block
stmt in catch block
java.lang.ArithmetricException: / by zero
stmt in finally block
```

User defined Exception

- In java we can create our own exception class and throw that exception using throw keyword.
- **Checked** – Extends `java.lang.Exception`, for recoverable condition, try-catch is compulsory to catch the exception explicitly, compile error.
- **Unchecked** – Extends `java.lang.RuntimeException`, for unrecoverable condition, like programming errors, no need try-catch, runtime error.
- These exceptions are known as **user-defined** or **custom** exceptions.
- The class name of your exception should end with *Exception*.

Custom checked exception:

- If the client is able to recover from the exception, make it a checked exception. To create a custom checked exception, extends `java.lang.Exception`
- For checked exception, you need to try and catch the exception.

Example:

```
class InvalidAgeException extends Exception

{

    InvalidAgeException(String s)

    {

        super(s);

    }

}

public class Main

{

    static void validate(int age) throws InvalidAgeException

    {

        if(age<18)

            throw new InvalidAgeException("not valid");

        else

            System.out.println("welcome to vote");

    }

    public static void main(String[] args) {

        System.out.println("Hello World");

        try

```

```

    {
        validate(13);

    }

    catch(Exception m)

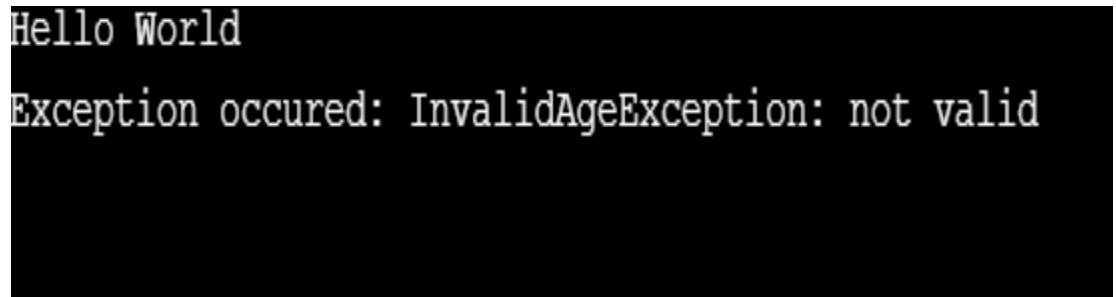
    {
        System.out.println("Exception occured: "+m);

    }

}

```

Output:



```
Hello World
Exception occured: InvalidAgeException: not valid
```

Custom unchecked exception:

- If the client cannot do anything to recover from the exception, make it an unchecked exception. To create a custom unchecked exception, extends `java.lang.RuntimeException`.
- For unchecked exception, try and catch the exception is optional.

Example:

```
import java.util.ArrayList;
```

```
import java.util.List;

class ListTooLargeException extends RuntimeException{

    public ListTooLargeException(String message) {
        super(message);
    }
}

public class Main
{
    public void analyze(List<String> data)
    {
        if (data.size() > 2) {
            //runtime exception
            throw new ListTooLargeException("List can't exceed 2 items!");
        }
    }

    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}

Main obj = new Main();
```

```

List<String> data = new ArrayList<String>();

Hello World
Exception in thread "main" ListTooLargeException: List can't exceed 2 items!
    at Main.analyze(Main.java:26)
    at Main.main(Main.java:37)

data.add("a");
    data.add("b");
    data.add("c");
    obj.analyze(data);
}

}

```

Output:

Details of the exception

- There are two ways to find the details of the exception, one is the printStackTrace() method and another is the getMessage() method.

[printStackTrace\(\) method](#)

- This is the method which is defined in java.lang.Throwable class and it is inherited into java.lang.Error class and java.lang.Exception class.
- This method will display the name of the exception and nature of the message and line number where an exception has occurred.

Example:

```

public class PrintStackTraceMethod {
    public static void main(String[] args) {
        try {
            int a[] = new int[5];

```

```
a[5]=20;

} catch (Exception e) {
    e.printStackTrace();
}

}

}
```

Output:

```
java.lang.ArrayIndexOutOfBoundsException: 5
        at PrintStackTraceMethod.main(PrintStackTraceMethod.java:5)
```

getMessage() method

- This is a method which is defined in `java.lang.Throwable` class and it is inherited into `java.lang.Error` and `java.lang.Exception` classes.
- This method will display the only exception message.

Example:

```
public class GetMessageMethod {
    public static void main(String[] args) {
        try {
            int x=1/0;
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Output:

```
/ by zero
```

Reference links:

<https://beginnersbook.com/2013/04/java-exception-handling/>

<https://www.geeksforgeeks.org/exceptions-in-java/>

<https://www.geeksforgeeks.org/throw-throws-java/>

<https://beginnersbook.com/2013/04/java-finally-block/>

<https://stackify.com/java-custom-exceptions/>

<https://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html>

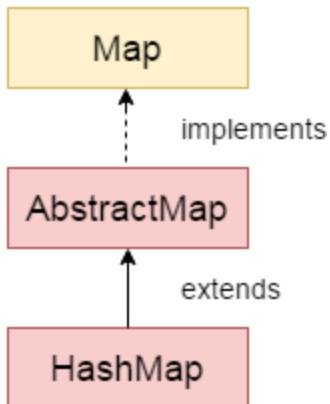
<https://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html>

Java HashMap class

Java HashMap class implements the map interface by using a hash table. It inherits AbstractMap class and implements Map interface.

Points to remember

- Java HashMap class contains values based on the key.
- Java HashMap class contains only unique keys.
- Java HashMap class may have one null key and multiple null values.
- Java HashMap class is non synchronized.
- Java HashMap class maintains no order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.



Working of HashMap in Java

What is Hashing

It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.

What is HashMap

`HashMap` is a part of the Java collection framework. It uses a technique called Hashing. It implements the map interface. It stores the data in the pair of Key and Value. `HashMap` contains an array of the nodes, and the node is represented as a class. It uses an array and `LinkedList` data structure internally for storing Key and Value. There are four fields in `HashMap`.

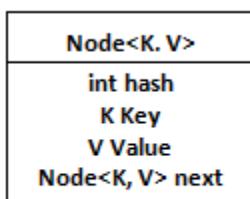


Figure: Representation of a Node

Before understanding the internal working of `HashMap`, you must be aware of `hashCode()` and `equals()` method.

- **equals():** It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.
- **hashCode():** This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.
- **Buckets:** Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.

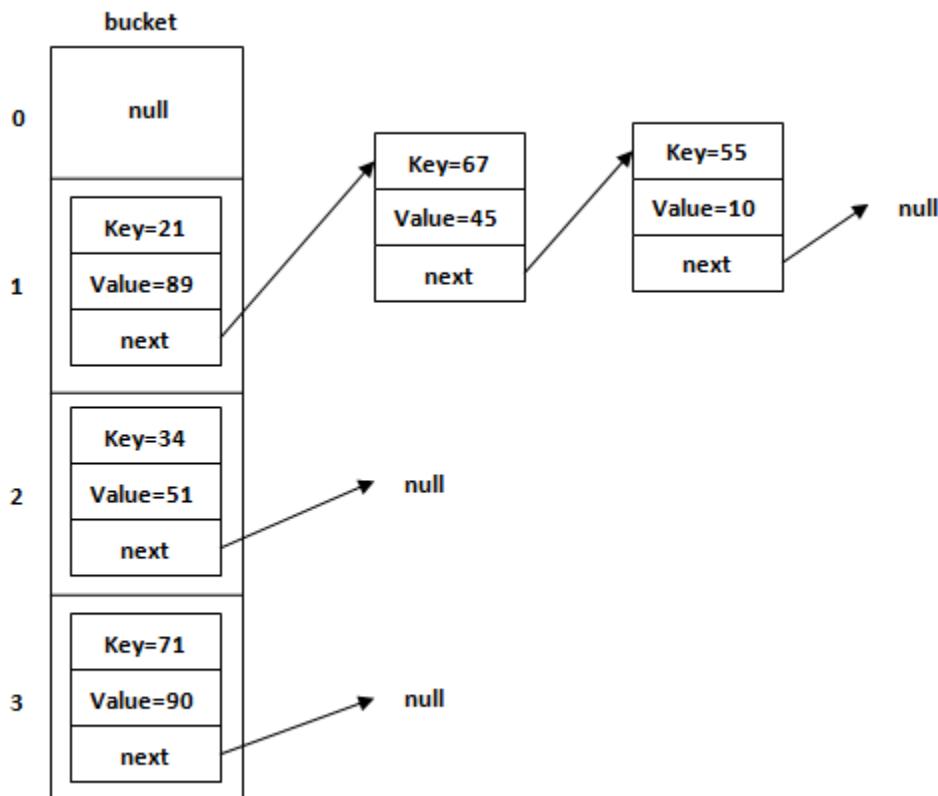


Figure: Allocation of nodes in Bucket

Insert Key, Value pair in HashMap

We use put() method to insert the Key and Value pair in the HashMap. The default size of HashMap is 16 (0 to 15).

Example

In the following example, we want to insert three (Key, Value) pair in the HashMap.

1. `HashMap<String, Integer> map = new HashMap<>();`
2. `map.put("Aman", 19);`
3. `map.put("Sunny", 29);`
4. `map.put("Ritesh", 39);`

Let's see at which index the Key, value pair will be saved into HashMap. When we call the `put()` method, then it calculates the hash code of the Key "Aman." Suppose the hash code of "Aman" is 2657860. To store the Key in memory, we have to calculate the index.

Calculating Index

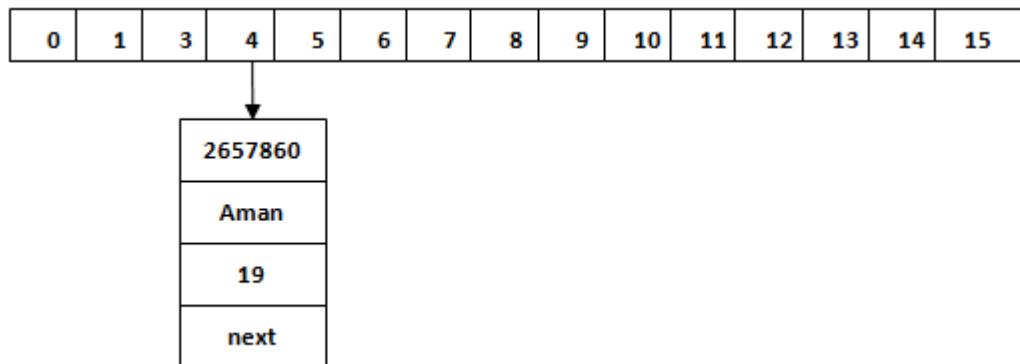
Index minimizes the size of the array. The Formula for calculating the index is:

1. $\text{Index} = \text{hashcode}(\text{Key}) \& (n-1)$

Where n is the size of the array. Hence the index value for "Aman" is:

1. $\text{Index} = 2657860 \& (16-1) = 4$

The value 4 is the computed index value where the Key and value will store in HashMap.



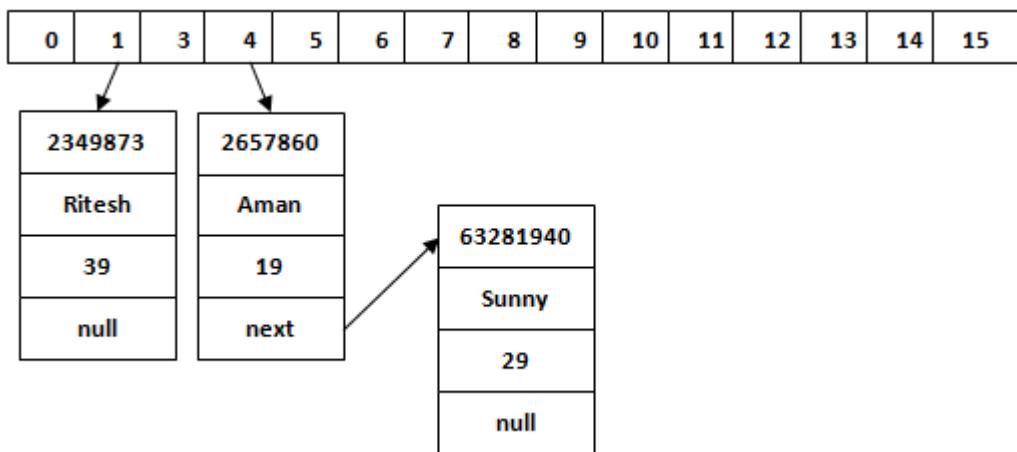
Hash Collision

This is the case when the calculated index value is the same for two or more Keys. Let's calculate the hash code for another Key "Sunny." Suppose the hash code for "Sunny" is

63281940. To store the Key in the memory, we have to calculate index by using the index formula Index=63281940 & (16-1) = 4

The value 4 is the computed index value where the Key will be stored in HashMap. In this case, equals() method check that both Keys are equal or not. If Keys are same, replace the value with the current value. Otherwise, connect this node object to the existing node object through the LinkedList. Hence both Keys will be stored at index 4.

Similarly, we will store the Key "Ritesh." Suppose hash code for the Key is 2349873. The index value will be 1. Hence this Key will be stored at index 1.



get() method in HashMap

get() method is used to get the value by its Key. It will not fetch the value if you don't know the Key. When get(K Key) method is called, it calculates the hash code of the Key.

Suppose we have to fetch the Key "Aman." The following method will be called.

1. map.get(**new** Key("Aman"));

It generates the hash code as 2657860. Now calculate the index value of 2657860 by using index formula. The index value will be 4, as we have calculated above. get() method search for the index value 4. It compares the first element Key with the given Key. If both keys are equal, then it returns the value else check for the next element in the node if it exists. In our scenario, it is found as the first element of the node and return the value 19.

Let's fetch another Key "Sunny."

The hash code of the Key "Sunny" is 63281940. The calculated index value of 63281940 is 4, as we have calculated for put() method. Go to index 4 of the array and compare the first element's Key with the given Key. It also compares Keys. In our scenario, the given Key is the second element, and the next of the node is null. It compares the second element Key with the specified Key and returns the value 29. It returns null if the next of the node is null.

Methods in HashMap

1. **void clear():** Used to remove all mappings from a map.
2. **boolean containsKey(Object key):** Used to return True if for a specified key, mapping is present in the map.
3. **boolean containsValue(Object value):** Used to return true if one or more key is mapped to a specified value.
4. **Object clone():** It is used to return a shallow copy of the mentioned hash map.
5. **boolean isEmpty():** Used to check whether the map is empty or not. Returns true if the map is empty.
6. **Set entrySet():** It is used to return a set view of the hash map.
7. **Object get(Object key):** It is used to retrieve or fetch the value mapped by a particular key.
8. **Set keySet():** It is used to return a set view of the keys.
9. **int size():** It is used to return the size of a map.
10. **Object put(Object key, Object value):** It is used to insert a particular mapping of key-value pair into a map.
11. **putAll(Map M):** It is used to copy all of the elements from one map into another.

12. **Object remove(Object key):** It is used to remove the values for any particular key in the Map.
13. **Collection values():** It is used to return a Collection view of the values in the HashMap.
14. **compute(K key, BiFunction<K, V> remappingFunction):** This method Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
15. **computeIfAbsent(K key, Function<K> mappingFunction):** This method If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
16. **computeIfPresent(K key, BiFunction<K, V> remappingFunction):** This method If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
17. **forEach(BiConsumer<K, V> action):** This method Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
18. **getOrDefault(Object key, V defaultValue):** This method returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
19. **merge(K key, V value, BiFunction<K, V> remappingFunction):** This method If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
20. **putIfAbsent(K key, V value):** This method If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
21. **replace(K key, V value):** This method replaces the entry for the specified key only if it is currently mapped to some value.
22. **replace(K key, V oldValue, V newValue):** This method replaces the entry for the specified key only if currently mapped to the specified value.
23. **replaceAll(BiFunction<K, V> function):** This method replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

REFERENCE LINK:

<https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/>

HashSet vs. TreeSet vs. LinkedHashSet

HashSet is Implemented using a hash table. Elements are not ordered.
The add, remove, and contains methods have constant time complexity O(1).

TreeSet is implemented using a tree structure(red-black tree in algorithm book).
The elements in a set are sorted, but the add, remove, and contains methods

has time complexity of $O(\log(n))$. It offers several methods to deal with the ordered set like `first()`, `last()`, `headSet()`, `tailSet()`, etc.

`LinkedHashSet` is between `HashSet` and `TreeSet`. It is implemented as a hash table with a linked list running through it, so it provides the order of insertion. The time complexity of basic methods is $O(1)$.

HASHTABLE IN JAVA

This class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value. It inherits `Dictionary` class and implements the `Map` interface.

To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the `hashCode` method and the `equals` method.

Points to remember

- A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the `hashcode()` method. A Hashtable contains values based on the key.
- Java Hashtable class contains unique elements.
- Java Hashtable class doesn't allow null key or value.
- Java Hashtable class is synchronized.
- In Hashtable we specify an object that is used as a key, and the value we want to associate to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

Hashtable class declaration

Let's see the declaration for `java.util.Hashtable` class.

1. **public class** `Hashtable<K,V>` **extends** `Dictionary<K,V>` **implements** `Map<K,V>`,
`Cloneable`, `Serializable`

Constructors:

- **Hashtable()**: This is the default constructor.

- **Hashtable(int size):** This creates a hash table that has initial size specified by size.
- **Hashtable(int size, float fillRatio):** This version creates a hash table that has initial size specified by size and fill ratio specified by fillRatio. **fill ratio:** Basically it determines how full hash table can be before it is resized upward.and its Value lie between 0.0 to 1.0
- **Hashtable(Map m):** This creates a hash table that is initialised with the elements in m.

Method	Description
void clear()	It is used to reset the hash table.
Object clone()	It returns a shallow copy of the Hashtable.
V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null.
V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.
Enumeration elements()	It returns an enumeration of the values in the hash table.

<code>Set<Map.Entry<K,V>> entrySet()</code>	It returns a set view of the mappings contained in the map.
<code>boolean equals(Object o)</code>	It is used to compare the specified Object with the Map.
<code>void forEach(BiConsumer<? super K,? super V> action)</code>	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
<code>V getOrDefault(Object key, V defaultValue)</code>	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
<code>int hashCode()</code>	It returns the hash code value for the Map
<code>Enumeration<K> keys()</code>	It returns an enumeration of the keys in the hashtable.
<code>Set<K> keySet()</code>	It returns a Set view of the keys contained in the map.
<code>V merge(K key, V value, BiFunction<? super V,? super V, extends V> remappingFunction)</code>	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
<code>V put(K key, V value)</code>	It inserts the specified value with the specified key in the hash table.
<code>void putAll(Map<? extends K,? extends V> t))</code>	It is used to copy all the key-value pair from map to hashtable.
<code>V putIfAbsent(K key, V value)</code>	If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
<code>boolean remove(Object key, Object value)</code>	It removes the specified values with the associated specified keys from the hashtable.

V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.
void replaceAll(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
String toString()	It returns a string representation of the Hashtable object.
Collection values()	It returns a collection view of the values contained in the map.
boolean contains(Object value)	This method returns true if some value equal to the value exists within the hash table, else return false.
boolean containsValue(Object value)	This method returns true if some value equal to the value exists Within the hash table, else return false.
boolean containsKey(Object key)	This method return true if some key equal to the key exists within the hash table, else return false.
boolean isEmpty()	This method returns true if the hash table is empty; returns false if it contains at least one key.
protected void rehash()	It is used to increase the size of the hash table and rehashes all of its keys.
V get(Object key)	This method returns the object that contains the value associated with the key.
V remove(Object key)	It is used to remove the key and its value. This method returns the value associated with the key.

int size()	This method returns the number of entries in the hash table.
------------	--

Tests if some key maps into the specified value in this hashtable. This operation is more expensive than the containsKey method

Returns true if this hashtable maps one or more keys to this value.

// Java program to demonstrate
// compute(Key, BiFunction) method.

```
import java.util.*;  
  
public class GFG {  
  
    // Main method  
    public static void main(String[] args)  
    {  
  
        // Create a Map and add some values  
        Map<String, String> map = new HashMap<>();  
        map.put("Name", "Aman");  
        map.put("Address", "Kolkata");  
  
        // Print the map  
        System.out.println("Map: " + map);  
  
        // remap the values using compute() method  
        map.compute("Name", (key, val)  
                    -> val.concat(" Singh"));  
        map.compute("Address", (key, val)  
                    -> val.concat(" West-Bengal"));  
  
        // print new mapping  
        System.out.println("New Map: " + map);  
    }  
}
```

Output:

```
Map: {Address=Kolkata, Name=Aman}
```

```
New Map: {Address=Kolkata West-Bengal, Name=Aman Singh}
```

// Java program to demonstrate

```

// compute(Key, BiFunction) method.

import java.util.*;

public class GFG {

    // Main method
    public static void main(String[] args)
    {

        // Create a Map and add some values
        Map<String, Integer> map = new HashMap<>();
        map.put("Key1", 12);
        map.put("Key2", 15);

        // print map details
        System.out.println("Map: " + map);

        // remap the values
        // using compute method
        map.compute("Key1", (key, val)
                    -> (val == null)
                        ? 1
                        : val + 1);
        map.compute("Key2", (key, val)
                    -> (val == null)
                        ? 1
                        : val + 5);

        // print new mapping
        System.out.println("New Map: " + map);
    }
}

```

Output:

```
Map: {Key2=15, Key1=12}
```

```
New Map: {Key2=20, Key1=13}
```

```

// Java program to demonstrate Exception thrown by
// compute(Key, BiFunction) method.

import java.util.*;

public class GFG {

```

```

// Main method
public static void main(String[] args)
{
    // create a Map and add some values
    Map<String, Integer> map = new HashMap<>();
    map.put("Key1", 12);
    map.put("Key2", 15);

    // print map details
    System.out.println("Map: " + map);

    try {
        // remap the values using compute() method
        // passing null value will throw exception
        map.compute(null, (key, value)
                    -> value + 3);
        System.out.println("New Map: " + map);
    }
    catch (NullPointerException e) {
        System.out.println("Exception: " + e);
    }
}

```

Output:

```

Map: {Key2=15, Key1=12}
Exception: java.lang.NullPointerException

```

```

//Java program to demonstrate
// computeIfAbsent(Key, Function) method.
import java.util.*;

public class Main {
    public static void main(String[] args)
    {
        Map<String, Integer> table = new Hashtable<>();

```

```

        table.put("Pen", 10);
        table.put("Book", 500);
        table.put("Clothes", 400);
        table.put("Mobile", 5000);
        // print map details
        System.out.println("hashTable: "
                + table.toString());
        // provide value for new key which is absent
        // using computeIfAbsent method
        table.computeIfAbsent("newPen", k -> 600);
        table.computeIfAbsent("newBook", k -> 800);
        table.computeIfAbsent("Book", k->1000);
        // print new mapping
        System.out.println("new hashTable: "
                + table);
    }
}

```

Output:

```

hashTable: {Book=500, Mobile=5000, Pen=10, Clothes=400}

new hashTable: {newPen=600, Book=500, newBook=800, Mobile=5000, Pen=10
, Clothes=400}

```

//Example for computeIfPresent()

1. **import** java.util.concurrent.*;
2. **import** java.util.*;
3. **public class** ConcurrentHashMapcomputeIfPresentExample1 {
4. **public static void** main(String[] args)
5. {
6. HashMap<String, Integer> mapcon = **new** HashMap<>();

```

7.     mapcon.put("k1", 100);
8.     mapcon.put("k2", 200);
9.     mapcon.put("k3", 300);
10.    mapcon.put("k4", 400);
11.
12.    System.out.println("HashMap values :\n " + mapcon.toString());
13.
14.    mapcon.computeIfPresent("k4", (key , val) -> val + 100);
15.
16.    mapcon.computeIfPresent("k5", (key , val) -> val + 100);
17.
18.    System.out.println("New HashMap after computeIfPresent :\n " + mapc
on);
19. }
20. }
```

Output:

```

HashMap values :
{k1=100, k2=200, k3=300, k4=400}
New HashMap after computeIfPresent :
{k1=100, k2=200, k3=300, k4=500}
```

Java forEach example using Map

We already saw above program to iterate over all entries of a [HashMap](#) and perform an action.

We can also iterate over map keys and values and perform any action on all elements.

Java 8 forEach map entries

```
HashMap<String, Integer> map = new HashMap<>();
```

```

map.put("A", 1);
map.put("B", 2);
map.put("C", 3);
```

```
//1. Map entries  
Consumer<Map.Entry<String, Integer>> action = System.out::println;
```

```
map.entrySet().forEach(action);
```

```
//2. Map keys
```

```
Consumer<String> actionOnKeys = System.out::println;
```

```
map.keySet().forEach(actionOnKeys);
```

```
//3. Map values
```

```
Consumer<Integer> actionOnValues = System.out::println;
```

```
map.values().forEach(actionOnValues);
```

Program output.

A=1

B=2

C=3

A

B

C

1

2

3

Reference Link:

<https://howtodoinjava.com/java8/foreach-method-example/>

```
//Example for getOrDefault()

// Java program to demonstrate
// getOrDefault(Object key, V defaultValue) method

import java.util.*;

public class GFG {

    // Main method
    public static void main(String[] args)
    {

        // Create a HashMap and add some values
        HashMap<String, Integer> map
            = new HashMap<>();
        map.put("a", 100);
        map.put("b", 200);
        map.put("c", 300);
        map.put("d", 400);

        // print map details
        System.out.println("HashMap: "
            + map.toString());

        // provide key whose value has to be obtained
        // and default value for the key. Store the
        // return value in k
        int k = map.getOrDefault("b", 500);
        int j= map.getOrDefault("y",100);

        // print the value of k returned by
        // getOrDefault(Object key, V defaultValue) method
        System.out.println("Returned Value for k : " + k);
        System.out.println("returned Valuefor j :" +j);
    }
}
```

Output:

HashMap: {a=100, b=200, c=300, d=400}

Returned Value for k: 200

Returned Value for j:100

//Example for Merge():

```
import java.util.HashMap;  
  
import java.util.Map;  
  
import java.util.function.BiFunction;  
  
public class TestMap4 {  
  
    public static void main(String[] args) {  
  
        Map<Integer, Integer> map = new HashMap<Integer, Integer>();  
  
        map.put(1, 1);  
  
        map.put(2, 2);  
  
        map.put(3, null);  
  
        System.out.println(map); // {1=1, 2=2, 3=null}  
  
        map.merge(1, 10, (x, y) -> x + y);  
  
        // key 1 is present, so new value 10 will be added to previous value 1  
  
        System.out.println(map); // {1=11, 2=2, 3=null}  
  
        map.merge(2, 10, (x, y) -> x < y ? x : y);  
  
        // Previous value for key=2 is less than new value. So, the old value  
        // remains as per the BiFunction  
  
        System.out.println(map); // {1=11, 2=2, 3=null}  
  
        map.merge(3, 10, (x, y) -> x * y);  
  
        // The old value for key=3 is null . But its not null * 3, the value 10
```

```

// will be added for the key

System.out.println(map); // {1=11, 2=2, 3=10}

map.merge(4, 10, (x, y) -> x / y);

// key=4 is not in map. So, BiFunction not evaluated, the element is

// added to map

System.out.println(map); // {1=11, 2=2, 3=10, 4=10}

map.merge(1, 10, (x, y) -> null);

// Since the BiFunction results in null, the element will be removed

// from map

System.out.println(map); // {2=2, 3=10, 4=10}

}

}

```

Output :

```

{1=1, 2=2, 3=null}
{1=11, 2=2, 3=null}
{1=11, 2=2, 3=null}
{1=11, 2=2, 3=10}
{1=11, 2=2, 3=10, 4=10}
{2=2, 3=10, 4=10}

```

//Example for replaceAll()

```

import java.util.*;

public class GFG {

    // Main method
    public static void main(String[] args)

```

```

{
    // create a HashMap having some entries
    HashMap<String, Integer>
        map1 = new HashMap<>();
    map1.put("key1", 1);
    map1.put("key2", 2);
    map1.put("key3", 3);
    map1.put("key4", 4);

    // print map details
    System.out.println("HashMap1: "
        + map1.toString());

    // replace oldValue with square of oldValue
    // using replaceAll method
    map1.replaceAll((key, oldValue)
        -> oldValue * oldValue);

    // print new mapping
    System.out.println("New HashMap: "
        + map1);
}
}

```

Output:

HashMap1: {key1=1, key2=2, key3=3, key4=4}

New HashMap: {key1=1, key2=4, key3=9, key4=16}

Hashtable elements() Method in Java

The `java.util.Hashtable.elements()` method of `Hashtable` class in Java is used to get the enumeration of the values present in the hashtable.

Syntax:

`Enumeration enum = Hash_table.elements()`

Parameters: The method does not take any parameters.

```
// Java code to illustrate the elements() method
import java.util.*;
```

```
public class Hash_Table_Demo {
```

```

public static void main(String[] args)
{
    // Creating an empty Hashtable
    Hashtable<Integer, String> hash_table =
        new Hashtable<Integer, String>();

    // Inserting elements into the table
    hash_table.put(10, "Geeks");
    hash_table.put(15, "4");
    hash_table.put(20, "Geeks");
    hash_table.put(25, "Welcomes");
    hash_table.put(30, "You");

    // Displaying the Hashtable
    System.out.println("The Table is: " + hash_table);

    // Creating an empty enumeration to store
    Enumeration enu = hash_table.elements();

    System.out.println("The enumeration of values are:");

    // Displaying the Enumeration
    while (enu.hasMoreElements()) {
        System.out.println(enu.nextElement());
    }
}

```

Output:

The Table is: {10=Geeks, 20=Geeks, 30=You, 15=4, 25=Welcomes}

The enumeration of values are:

Geeks

Geeks

You

4

Welcomes

entrySet() : used to get a set view of the entries contained in this hash table.

Syntax :public Set<Map.Entry> entrySet()

Returns :returns a set view of the mappings contained in this map.

Exception :NA

```
// Java code illustrating entrySet() method
import java.util.*;
class hashTabledemo {
    public static void main(String[] args)
    {
        // creating a hash table
        Hashtable<Integer, String> h =
            new Hashtable<Integer, String>();

        h.put(3, "Geeks");
        h.put(2, "forGeeks");
        h.put(1, "isBest");

        // creating set view for hash table
        Set s = h.entrySet();

        // printing set entries
        System.out.println("set entries: " + s);
    }
}
```

Output:

```
set entries: [3=Geeks, 2=forGeeks, 1=isBest]
```

boolean equals(Object o) : used to compare specified object with this Map for equality.

Syntax :public boolean equals(Object o)

Returns :returns true if the specified Object is equal to this Map.

Exception :NA

```
// Java code illustrating equal() method
import java.util.*;
class hashTabledemo {
    public static void main(String[] args)
    {
        // creating a hash table
        Hashtable<Integer, String> h =
            new Hashtable<Integer, String>();

        Hashtable<Integer, String> h1 =
            new Hashtable<Integer, String>();
```

```

h.put(3, "Geeks");
h.put(2, "forGeeks");
h.put(1, "isBest");

h1.put(3, "Geeks");
h1.put(2, "forGeeks");
h1.put(1, "isBest");

// checking whether both hash tables
// are equal or not
if (h.equals(h1))
    System.out.println("both are equal");
}
}

```

Output:

both are equal

1. **Object get(Object key)** : used to get the object that contains the value associated with key.
2. **Syntax** :public Object get(Object key)
3. **Returns** :the value to which the key is mapped in this hashtable.
4. **Exception** :NullPointerException if the key is null.

```

// Java code illustrating get() method
import java.util.*;
class Vector_demo {
    public static void main(String[] args)
    {

        // creating a hash table
        Hashtable<String, Integer> marks =
            new Hashtable<String, Integer>();

        // enter name/marks pair
        marks.put("tweener", new Integer(345));
        marks.put("krantz", new Integer(245));
        marks.put("burrows", new Integer(790));
        marks.put("tancredi", new Integer(365));
    }
}
```

```
marks.put("bellick", new Integer(435));

// get the value mapped with key krantz

System.out.println(marks.get("krantz"));
}
}
```

Output:

245

5. **int hashCode()** :returns the hash code value for this Map as per the definition in the Map interface.
6. **Syntax** :public int hashCode()
7. **Returns** :a hash code value for this object.
8. **Exception** :NA

```
// Java code illustrating hashCode() method
import java.util.*;
class hashTabledemo {
    public static void main(String[] args)
    {
        // creating a hash table
        Hashtable<Integer, String> h =
            new Hashtable<Integer, String>();

        h.put(3, "Geeks");
        h.put(2, "forGeeks");
        h.put(1, "isBest");

        // obtaining hash code
        System.out.println("hash code is: " + h.hashCode());
    }
}
```

Output:

hash code is: -672864097

9. **boolean isEmpty()** :used to test if this hashtable maps no keys to values.
10. **Syntax** :public boolean isEmpty()
11. **Returns** :true if this hashtable maps no keys to values; false otherwise.
12. **Exception** :NA

```
// Java code illustrating isEmpty() method
```

```

import java.util.*;
class hashTabledemo {
    public static void main(String[] arg)
    {
        // creating a hash table
        Hashtable<Integer, String> h =
            new Hashtable<Integer, String>();

        h.put(3, "Geeks");
        h.put(2, "forGeeks");
        h.put(1, "isBest");

        // clear hash table h
        h.clear();

        // checking whether hash table h is empty or not
        if (h.isEmpty())
            System.out.println("yes hash table is empty");
    }
}

```

Output:

yes hash table is empty

13. **Enumeration keys()** :used to get enumeration of the keys contained in the hash table.
14. **Syntax** :public Enumeration keys()
15. **Returns** :an enumeration of the keys in this hashtable.
16. **Exception** :NA

```

// Java code illustrating keys() method
import java.util.*;
class hashTabledemo {
    public static void main(String[] arg)
    {
        // creating a hash table
        Hashtable<Integer, String> h =
            new Hashtable<Integer, String>();

        Hashtabl<Integer, String>e h1 =
            new Hashtable<Integer, String>();

        h.put(3, "Geeks");

```

```

        h.put(2, "forGeeks");
        h.put(1, "isBest");

        // create enumeration

        Enumeration e1 = h.keys();
        System.out.println("display key:");

        while (e1.hasMoreElements()) {
            System.out.println(e1.nextElement());
        }
    }
}

```

Output:

```

display key:
3
2
1

```

17. **Object put(Object key, Object value)** :maps the specified key to the specified value in this hashtable.
18. **Syntax** :public Object put(Object key, Object value)
19. **Returns** : returns null if key is not already in the hash table;
20. returns the previous value associated with key if key is already in the hash table.
21. **Exception** :NullPointerException if the key or value is null.

```

// Java code illustrating put() method
class hashTabledemo {
    public static void main(String[] arg)
    {
        // creating a hash table
        Hashtable<Integer, String> h =
            new Hashtable<Integer, String>();

        // key/value pair
        h.put(3, "Geeks");
        h.put(2, "forGeeks");
        h.put(1, "isBest");

        System.out.println("entries in table: " + h);
    }
}

```

```
    }  
}
```

Output:

```
entries in table: {3=Geeks, 2=forGeeks, 1=isBest}
```

22. **putIfAbsent(Key, Function):** The putIfAbsent(Key, value) method of Hashtable class which allows to map a value to a given key if given key is not associated with a value or mapped to null. A null value is returned if such key-value set is already present in the HashMap.
23. **Syntax:** public <K, V>
 computefAbsent(K key, V value)
- 25.
26. **Parameters:** This method accepts two parameters:
 - key: specifies the key to which the specified value is to be mapped if key is not associated with any value.
27.
 - value: specifies the value to be mapped to the specified key.
- 28.
- 29.
30. **Returns:** This method returns the existing value mapped to the key and returns null if no value is previously mapped to the key.
- 31.
32. **Exception:** This method throws **NullPointerException** when the specified parameters are null.

Below programs illustrate the putIfAbsent(Key, Value) method:

```
// Java program to demonstrate  
// putIfAbsent(key, value) method.  
  
import java.util.*;  
  
public class GFG {  
  
    // Main method  
    public static void main(String[] args)  
    {  
  
        // create a table and add some values  
        Map<String, Integer>  
        table = new Hashtable<>();  
  
        table.put("Pen", 10);
```

```

table.put("Book", 500);
table.put("Clothes", 400);
table.put("Mobile", 5000);

// print map details
System.out.println("hashTable: "
+ table.toString());

// Inserting non-existing key with value
// using putIfAbsent method
String retValue
= String.valueOf(table
.putIfAbsent("Booklet", 2500));

// Print the returned value
System.out.println("Returned value "
+ "for Key 'Booklet' is: "
+ retValue);

// print new mapping
System.out.println("hashTable: "
+ table);

// Inserting existing key with value
// using putIfAbsent method
retValue
= String.valueOf(table
.putIfAbsent("Book", 4500));

// Print the returned value
System.out.println("Returned value"
+ " for key 'Book' is: "
+ retValue);

// print new mapping
System.out.println("hashTable: "
+ table);
}
}

```

Output:

hashTable: {Book=500, Mobile=5000, Pen=10, Clothes=400}

Returned value for Key 'Booklet' is: null

```
hashTable: {Book=500, Mobile=5000, Pen=10, Clothes=400, Booklet=2500}
Returned value for key 'Book' is: 500
hashTable: {Book=500, Mobile=5000, Pen=10, Clothes=400, Booklet=2500}
```

- 33. **KeySet()** :used to get a Set view of the keys contained in this hash table.
- 34. **Syntax** :public Set keySet()
- 35. **Returns** :a set view of the keys contained in this map.
- 36. **Exception** :NA

```
// Java code illustrating keySet() method
import java.util.*;
class hashTabledemo {
    public static void main(String[] args)
    {
        // creating a hash table
        Hashtable<Integer, String> h =
            new Hashtable<Integer, String>();

        h.put(3, "Geeks");
        h.put(2, "forGeeks");
        h.put(1, "isBest");

        // creating set view for keys
        Set sKey = h.keySet();

        // checking key set
        System.out.println("key set: " + sKey);
    }
}
```

Output:

```
key set: [3, 2, 1]
```

- 37. **void putAll(Map t)** : copies all of the mappings from the specified map to this hashtable.
- 38. **Syntax** :public void putAll(Map t)
- 39. **Returns** :NA
- 40. **Exception** :NullPointerException if the specified map is null.

```
// Java code illustrating putAll() method
import java.util.*;
class hashTabledemo {
```

```

public static void main(String[] args)
{
    // creating a hash table
    Hashtable<Integer, String> h =
        new Hashtable<Integer, String>();

    Hashtable<Integer, String> h1 =
        new Hashtable<Integer, String>();

    h.put(3, "Geeks");
    h.put(2, "forGeeks");
    h.put(1, "isBest");

    // copy all element of h into h1
    h1.putAll(h);

    // checking h1
    System.out.println("Values in h1: " + h1);
}

```

Output:

```

Values in h1: {3=Geeks, 2=forGeeks, 1=isBest}
</pre>

```

41. **protected void rehash()** : Increase the size of the hash table and rehashes all its keys.
42. **Syntax :**protected void rehash()
43. **Returns :**NA
44. **Exception :**NA

Hashtable Class rehash() method

- **rehash() method** is available in **java.util** package.
- **rehash() method** is used to extend the capacity and it is invoked implicitly if the number of keys limit exceeds hashtable capacity.
- **rehash() method** is a non-static method, it is accessible with the class object only and if we try to access the method with the class name then we will get an error.
- **rehash() method** does not throw an exception at the time of extending capacity.

Syntax:

```
public void rehash();
```

Parameter(s):

- It does not accept any parameter.

Return value:

The return type of the method is **void**, it returns nothing.

45. **Object remove(Object key)** :Removes key and its value.
46. **Syntax** :public Object remove(Object key)
47. **Returns** :returns the value associated with key. If key is not
48. in the hash table, a null object is returned.
49. **Exception** :NullPointerException if specified key is null.

```
// Java code illustrating remove() method
import java.util.*;
class hashTabledemo {
    public static void main(String[] args)
    {
        // creating a hash table
        Hashtable<Integer, String> h =
            new Hashtable<Integer, String>();

        h.put(3, "Geeks");
        h.put(2, "forGeeks");
        h.put(1, "isBest");

        // remove value for 2 from Hashtable h
        h.remove(2);

        // checking Hashtable h
        System.out.println("values after remove: " + h);
    }
}
```

Output:

```
values after remove: {3=Geeks, 1=isBest}
```

50. **int size()** :returns the number of entries in hash table.
51. **Syntax** :public int size()
52. **Returns** :returns the number of keys in this hashtable.
53. **Exception** :NA

```
// Java code illustrating size() method
import.java.util.*;
class hashTabledemo {
    public static void main(String[] arg)
    {
        // creating a hash table
        Hashtable<String, Integer> marks =
            new Hashtable<String, Integer>();

        // enter name/marks pair
        marks.put("tweener", new Integer(345));
        marks.put("krantz", new Integer(245));
        marks.put("burrows", new Integer(790));
        marks.put("tancredi", new Integer(365));
        marks.put("bellick", new Integer(435));

        // size of hash table
        System.out.println("Size is: " + marks.size());
    }
}
```

Output:

```
Size is: 5
```

54. **String toString()** :returns the string equivalent of a hash table.
55. **Syntax** :public String toString()
56. **Returns** :
57. **Exception** :NA

```
// Java code illustrating toString() method
import java.util.*;
class hashTabledemo {
    public static void main(String[] arg)
    {
        // creating a hash table
        Hashtable<Integer, String> h =
            new Hashtable<Integer, String>();
```

```

        h.put(3, "Geeks");
        h.put(2, "forGeeks");
        h.put(1, "isBest");

        // String equivalent of map
        System.out.println("string equivalent" +
            " of map: " + h.toString());
    }
}

```

Output:

```
string equivalent of map: {3=Geeks, 2=forGeeks, 1=isBest}
```

58. **values()** :used to get a Collection view of the values contained in this Hashtable.

59. **Syntax** :public Collection values()

60. **Returns** :returns a collection view of the values contained

61. in this map.

Exception :NA

```

// Java code illustrating values() method
import java.util.*;
class hashTabledemo {
    public static void main(String[] arg)
    {
        // creating a hash table
        Hashtable<Integer, String> h =
            new Hashtable<Integer, String>();

        h.put(3, "Geeks");
        h.put(2, "forGeeks");
        h.put(1, "isBest");
    }
}
```

// creating set view for hash table
Set s = h.entrySet();

// checking collection view of values
System.out.println("collection values: " + h.values());

```

    }
}
```

Output:

```
collection values: [Geeks, forGeeks, isBest]
```

Reference Link:

<https://www.geeksforgeeks.org/hashtable-in-java/>

Difference between HashMap and Hashtable

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys.

But there are many differences between HashMap and Hashtable classes that are given below.

HashMap	Hashtable
1) HashMap is non synchronized . It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is synchronized . It is thread-safe and can be shared with many threads.
2) HashMap allows one null key and multiple null values .	Hashtable doesn't allow any null key or value .
3) HashMap is fast .	Hashtable is slow .
4) We can make the HashMap as synchronized by calling this code <pre>Map m = Collections.synchronizedMap(hashMap);</pre>	Hashtable is internally synchronized and can't be unsynchronized.
5) HashMap is traversed by Iterator .	Hashtable is traversed by Enumerator and Iterator .
6) HashMap inherits AbstractMap class.	Hashtable inherits Dictionary class.

INHERITANCE:

The process by which one class acquires the properties(data members) and functionalities(methods) of another class is called **inheritance**. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

Child Class:

The class that extends the features of another class is known as child class, sub class or derived class.

Parent Class:

The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base class.

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

Inheritance allows us to reuse of code, it improves reusability in your java application.

Note: The biggest **advantage of Inheritance** is that the code that is already present in base class need not be rewritten in the child class.

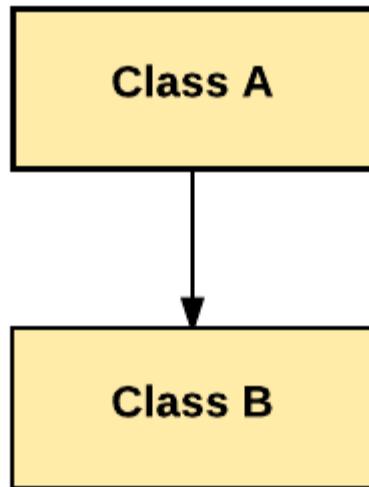
This means that the data members(instance variables) and methods of the parent class can be used in the child class as well.

Types of Inheritance

There are Various types of inheritance in Java:

Single Inheritance:

In Single Inheritance one class extends another class (one class only).

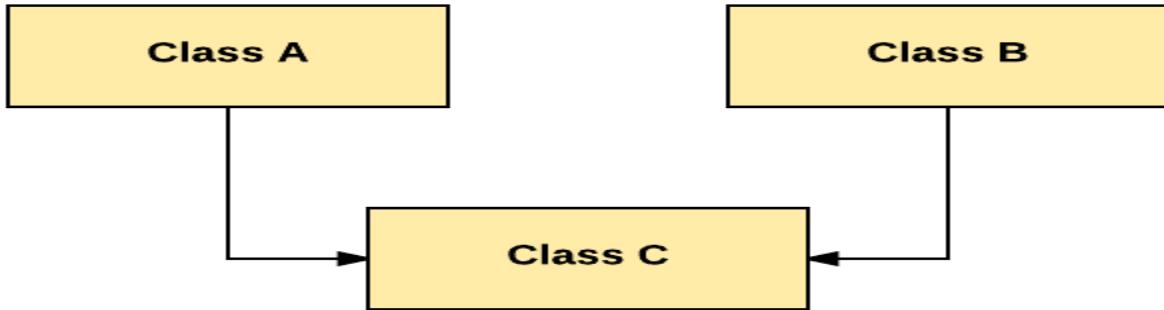


Single Inheritance

In above diagram, Class B extends only Class A. Class A is a super class and Class B is a Sub-class.

Multiple Inheritance:

In Multiple Inheritance, one class extending more than one class. Java does not support multiple inheritance.

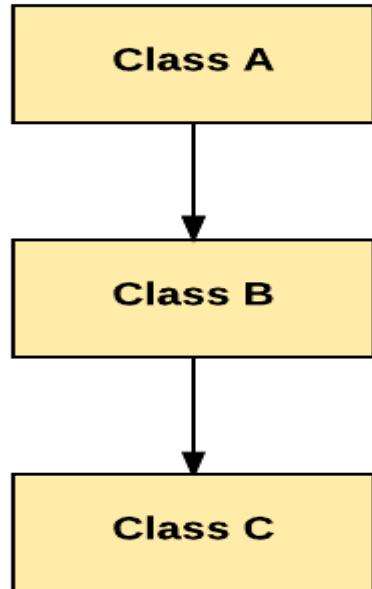


Multiple Inheritance

As per above diagram, Class C extends Class A and Class B both.

Multilevel Inheritance:

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

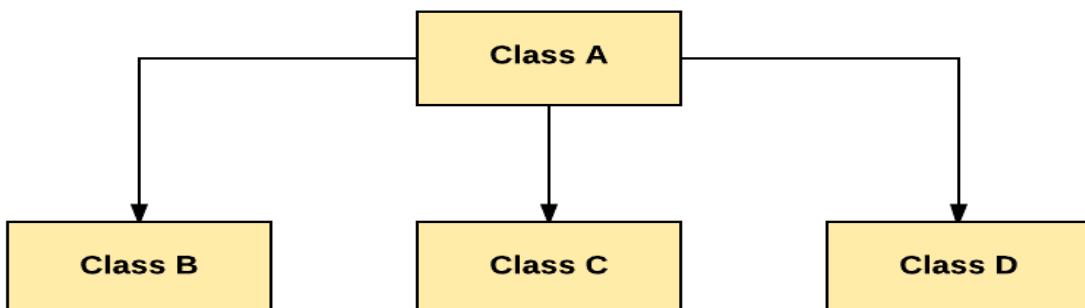


Multilevel Inheritance

As per shown in diagram Class C is subclass of B and B is a of subclass Class A.

Hierarchical Inheritance:

In Hierarchical Inheritance, one class is inherited by many sub classes.

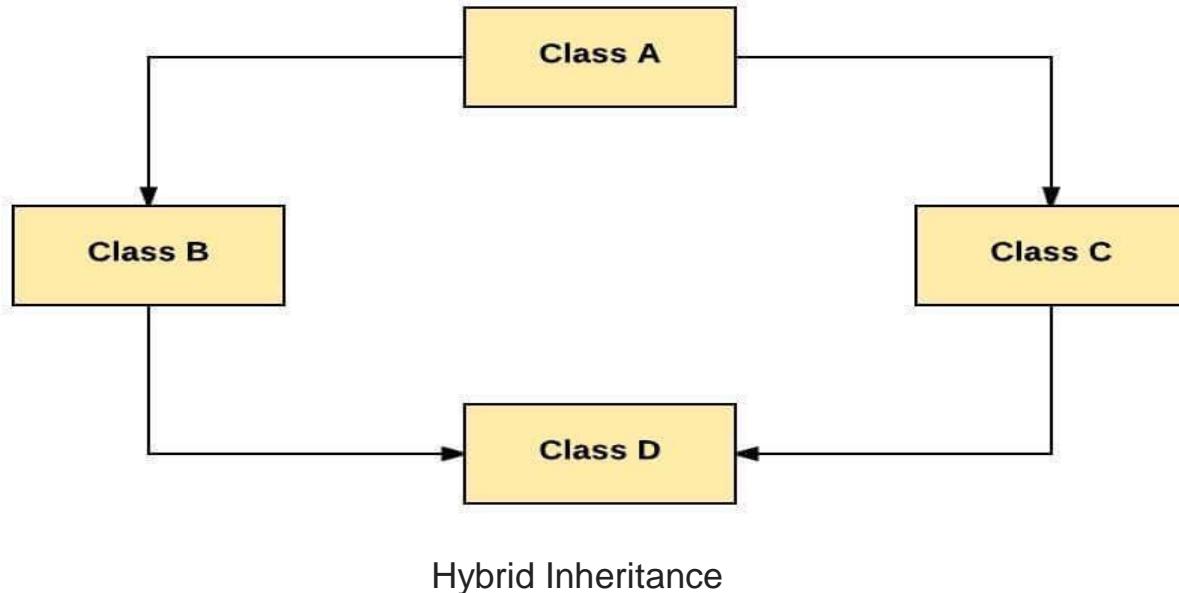


Hierarchical Inheritance

As per above example, Class B, C, and D inherit the same class A.

Hybrid Inheritance:

Hybrid inheritance is a combination of Single and Multiple inheritance.



As per above example, all the public and protected members of Class A are inherited into Class D, first via Class B and secondly via Class C.

Note: Java doesn't support hybrid/Multiple inheritance

Inheritance In Java

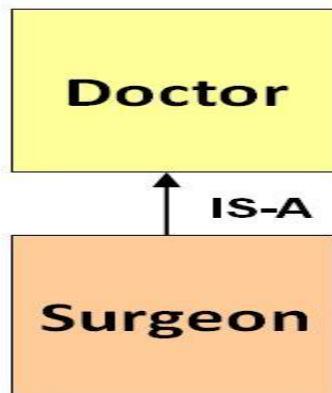
JAVA INHERITANCE is a mechanism in which one class acquires the property of another class. In Java, when an "Is-A" relationship exists between two classes, we use Inheritance. The parent class is called a super class and the inherited class is called a subclass. The keyword `extends` is used by the sub class to inherit the features of super class.

Inheritance is important since it leads to the reusability of code.

Java Inheritance Syntax:

```
class subClass extends superClass
{
    //methods and fields
}
```

Java Inheritance Example



```
class Doctor {  
    void Doctor_Details() {  
        System.out.println("Doctor Details...");  
    }  
}  
  
class Surgeon extends Doctor {  
    void Surgeon_Details() {  
        System.out.println("Surgeon Detail...");  
    }  
}  
  
public class Hospital {  
    public static void main(String args[]) {  
        Surgeon s = new Surgeon();  
        s.Doctor_Details();  
        s.Surgeon_Details();  
    }  
}
```

```
class Vehicle {  
  
    protected String brand = "Ford";      // Vehicle attribute  
  
    public void honk() {                  // Vehicle method  
        System.out.println("Tuut, tuut!");  
    }  
}
```

```
    }

}

class Car extends Vehicle {

    private String modelName = "Mustang"; // Car attribute

    public static void main(String[] args) {

        // Create a myCar object
        Car myCar = new Car();

        // Call the honk() method (from the Vehicle class) on the myCar object
        myCar.honk();

        // Display the value of the brand attribute (from the Vehicle class) and the
        // value of the modelName from the Car class
        System.out.println(myCar.brand + " " + myCar.modelName);
    }
}
```

Output:

```
Tuut, tuut!
Ford Mustang
```

```
class Teacher {
    String designation = "Teacher";
```

```

String collegeName = "Beginnersbook";
void does(){
    System.out.println("Teaching");
}
}

public class PhysicsTeacher extends Teacher{
    String mainSubject = "Physics";
    public static void main(String args[]){
        PhysicsTeacher obj = new PhysicsTeacher();
        System.out.println(obj.collegeName);
        System.out.println(obj.designation);
        System.out.println(obj.mainSubject);
        obj.does();
    }
}

```

Output:

```

Beginnersbook
Teacher
Physics
Teaching

```

Note:

The derived class inherits all the members and methods that are declared as public or protected. If the members or methods of super class are declared as private then the derived class cannot use them directly. The private members can be accessed only in its own class. Such private members can only be accessed using public or protected getter and setter methods of super class as shown in the example below.

```

class Teacher {
    private String designation = "Teacher";
    private String collegeName = "Beginnersbook";
    public String getDesignation() {
        return designation;
    }
    protected void setDesignation(String designation) {

```

```

        this.designation = designation;
    }
    protected String getCollegeName() {
        return collegeName;
    }
    protected void setCollegeName(String collegeName) {
        this.collegeName = collegeName;
    }
    void does(){
        System.out.println("Teaching");
    }
}

public class JavaExample extends Teacher{
    String mainSubject = "Physics";
    public static void main(String args[]){
        JavaExample obj = new JavaExample();
        /* Note: we are not accessing the data members
         * directly we are using public getter method
         * to access the private members of parent class
         */
        System.out.println(obj.getCollegeName());
        System.out.println(obj.getDesignation());
        System.out.println(obj.mainSubject);
        obj.does();
    }
}

```

The output is:

```

Beginnersbook
Teacher
Physics
Teaching

```

The important point to note in the above example is that the child class is able to access the private members of parent class through **protected methods** of parent class. When we make a instance variable(data member) or method **protected**, this means that they are accessible only in the class itself and in child class.

REFERENCE LINK: <https://beginnersbook.com/2013/03/inheritance-in-java/>

DEFAULT METHODS IN INTERFACE:

Prior to java 8, [interface in java](#) can only have abstract methods. All the methods of interfaces are public & abstract by default. Java 8 allows the interfaces to have default and static methods. The reason we have default methods in interfaces is to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.

Why default method?

For example, if several classes such as A, B, C and D implements an interface XYZInterface then if we add a new method to the XYZInterface, we have to change the code in all the classes(A, B, C and D) that implements this interface. In this example we have only four classes that implements the interface which we want to change but imagine if there are hundreds of classes implementing an interface then it would be almost impossible to change the code in all those classes. This is why in java 8, we have a new concept “default methods”. These methods can be added to any existing interface and we do not need to implement these methods in the implementation classes mandatorily, thus we can add these default methods to existing interfaces without breaking the code.

```
interface MyInterface{
    /* This is a default method so we need not
     * to implement this method in the implementation
     * classes
    */
    default void newMethod(){
        System.out.println("Newly added default method");
    }
    /* Already existing public and abstract method
     * We must need to implement this method in
     * implementation classes.
    */
    void existingMethod(String str);
}
public class Example implements MyInterface{
    // implementing abstract method
    public void existingMethod(String str){
        System.out.println("String is: "+str);
    }
    public static void main(String[] args) {
        Example obj = new Example();
```

```
//calling the default method of interface  
obj.newMethod();  
//calling the abstract method of interface  
obj.existingMethod("Java 8 is easy to learn");  
  
}  
}
```

Output:

```
Newly added default method  
String is: Java 8 is easy to learn
```

Static method in Interface

As mentioned above, the static methods in interface are similar to default method so we need not to implement them in the implementation classes. We can safely add them to the existing interfaces without changing the code in the implementation classes. Since these methods are static, we cannot override them in the implementation classes.

```
interface MyInterface{  
    /* This is a default method so we need not  
     * to implement this method in the implementation  
     * classes  
     */  
    default void newMethod(){  
        System.out.println("Newly added default method");  
    }  
  
    /* This is a static method. Static method in interface is  
     * similar to default method except that we cannot override  
     * them in the implementation classes.  
     * Similar to default methods, we need to implement these methods  
     * in implementation classes so we can safely add them to the  
     * existing interfaces.  
     */  
    static void anotherNewMethod(){  
        System.out.println("Newly added static method");  
    }  
    /* Already existing public and abstract method
```

```

        * We must need to implement this method in
        * implementation classes.
        */
void existingMethod(String str);
}
public class Example implements MyInterface{
    // implementing abstract method
    public void existingMethod(String str){
        System.out.println("String is: "+str);
    }
    public static void main(String[] args) {
        Example obj = new Example();

        //calling the default method of interface
        obj.newMethod();
        //calling the static method of interface
        MyInterface.anotherNewMethod();
        //calling the abstract method of interface
        obj.existingMethod("Java 8 is easy to learn");

    }
}

```

Output:

```

Newly added default method
Newly added static method
String is: Java 8 is easy to learn

```

REFERENCE LINK:

<https://beginnersbook.com/2017/10/java-8-interface-changes-default-method-and-static-method/>

<https://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method>

```

class ParentClass{
    //Parent class constructor
    ParentClass(){
        System.out.println("Constructor of Parent");
    }
    void disp(){
        System.out.println("Parent Method");
    }
}
class JavaExample extends ParentClass{
    JavaExample(){
        System.out.println("Constructor of Child");
    }
    void disp(){
        System.out.println("Child Method");
        //Calling the disp() method of parent class
        super.disp();
    }
    public static void main(String args[]){
        //Creating the object of child class
        JavaExample obj = new JavaExample();
        obj.disp();
    }
}

```

The output is :

```

Constructor of Parent
Constructor of Child
Child Method
Parent Method

```

Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC) is an [application programming interface](#) (API) for the programming language [Java](#), which defines how a client may access a [database](#). It is a Java-based data access technology used for Java database connectivity. JDBC stands

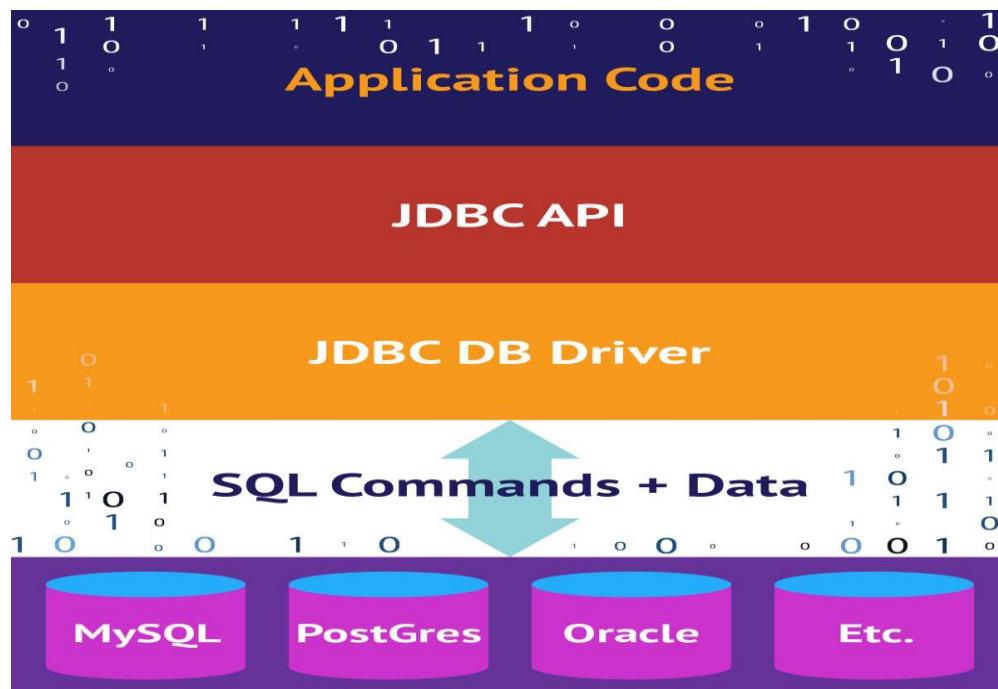
for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC interface consists of two layers:

1. The JDBC API supports communication between the Java application and the JDBC manager.
2. The JDBC driver supports communication between the JDBC manager and the database driver.

JDBC is the common API that your application code interacts with. Beneath that is the JDBC-compliant driver for the database you are using.

Figure 1 is an architectural overview of JDBC in the Java persistence layer.



Fundamental Steps in JDBC

The fundamental steps involved in the process of connecting to a database and executing a query consist of the following:

- Import JDBC packages.
- Load and register the JDBC driver.
- Open a connection to the database.
- Create a statement object to perform a query.

- Execute the statement object and return a query resultset.
- Process the resultset.
- Close the resultset and statement objects.
- Close the connection.

Import JDBC Packages

The **Import** statements tell the Java compiler where to find the classes you reference in your code and are placed at the very beginning of your source code.

To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code –

```
import java.sql.* ; // for standard JDBC programs
import java.math.* ; // for BigDecimal and BigInteger support
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.ResultSet;
import java.sql.Statement;
```

Each of these imports provides access to a class that facilitates the standard Java database connection:

- `Connection` represents the connection to the database.
- `DriverManager` obtains the connection to the database. (Another option is `DataSource`, used for connection pooling.)
- `SQLException` handles SQL errors between the Java application and the database.
- `ResultSet` and `Statement` model the data result sets and SQL statements.

Register JDBC Driver

You must register the driver in your program before you use it. Registering the driver is the process by which the Oracle driver's class file is loaded into the memory, so it can be utilized as an implementation of the JDBC interfaces.

You need to do this registration only once in your program. You can register a driver in one of two ways.

Approach I - Class.forName()

The most common approach to register a driver is to use Java's **Class.forName()** method, to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

The following example uses `Class.forName()` to register the Oracle driver –

```
try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
    System.out.println("Error: unable to load driver class!");
    System.exit(1);
}
```

Approach II - DriverManager.registerDriver()

The second approach you can use to register a driver, is to use the static **DriverManager.registerDriver()** method.

You should use the *registerDriver()* method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

The following example uses `registerDriver()` to register the Oracle driver –

```
try {
    Driver myDriver = new oracle.jdbc.driver.OracleDriver();
    DriverManager.registerDriver( myDriver );
}
catch(ClassNotFoundException ex) {
    System.out.println("Error: unable to load driver class!");
    System.exit(1);
}
```

Create Connection Object

We have listed down three forms of **DriverManager.getConnection()** method to create a connection object.

Using a Database URL with a username and password

The most commonly used form of `getConnection()` requires you to pass a database URL, a *username*, and a *password*:

Assuming you are using Oracle's **thin** driver, you'll specify a host:port:databaseName value for the database portion of the URL.

If you have a host at TCP/IP address 192.0.0.1 with a host name of amrood, and your Oracle listener is configured to listen on port 1521, and your database name is EMP, then complete database URL would be –

```
jdbc:oracle:thin:@amrood:1521:EMP
```

Now you have to call `getConnection()` method with appropriate username and password to get a **Connection** object as follows –

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password";
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

Database URL Formulation

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method. For easy reference, let me list the three overloaded `DriverManager.getConnection()` methods –

- `getConnection(String url)`
- `getConnection(String url, Properties prop)`
- `getConnection(String url, String user, String password)`

Here each form requires a database **URL**. A database URL is an address that points to your database.

Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	<code>com.mysql.jdbc.Driver</code>	<code>jdbc:mysql://hostname/ databaseName</code>
ORACLE	<code>oracle.jdbc.driver.OracleDriver</code>	<code>jdbc:oracle:thin:@hostname:port Number:databaseName</code>

DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname: port Number/databaseName

All the highlighted part in URL format is static and you need to change only the remaining part as per your database setup.

Using Only a Database URL

A second form of the `DriverManager.getConnection()` method requires only a database URL –

```
DriverManager.getConnection(String url);
```

However, in this case, the database URL includes the username and password and has the following general form –

```
jdbc:oracle:driver:username/password@database
```

So, the above connection can be created as follows –

```
String URL = "jdbc:oracle:thin:username/password@amrood:1521:EMP";
Connection conn = DriverManager.getConnection(URL);
```

Using a Database URL and a Properties Object

A third form of the `DriverManager.getConnection()` method requires a database URL and a `Properties` object –

```
DriverManager.getConnection(String url, Properties info);
```

A `Properties` object holds a set of keyword-value pairs. It is used to pass driver properties to the driver during a call to the `getConnection()` method.

To make the same connection made by the previous examples, use the following code –

```
import java.util.*;

String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
Properties info = new Properties();
info.put("user", "username");
info.put("password", "password");

Connection conn = DriverManager.getConnection(URL, info);
```

Create a statement

Once a connection is established you can interact with the database. The JDBCStatement, CallableStatement, and PreparedStatement interfaces define the methods that enable you to send SQL commands and receive data from your database. Use of JDBC Statement is as follows:

```
Statement st = con.createStatement();
```

Here, con is a reference to Connection interface used in previous step .

4. Execute the query

Now comes the most important part i.e executing the query. Query here is an SQL Query . Now we know we can have multiple types of queries. Some of them are as follows:

- Query for updating / inserting table in a database.
- Query for retrieving data .

The executeQuery() method of Statement interface is used to execute queries of retrieving values from the database. This method returns the object of ResultSet that can be used to get all the records of a table.

The executeUpdate(sql query) method of Statement interface is used to execute queries of updating/inserting .

Example:

```
int m = st.executeUpdate(sql);
if (m==1)
    System.out.println("inserted successfully : "+sql);
else
    System.out.println("insertion failed");
```

Closing JDBC Connections

At the end of your JDBC program, it is required explicitly to close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

Relying on the garbage collection, especially in database programming, is a very poor programming practice. You should make a habit of always closing the connection with the close() method associated with connection object.

To ensure that a connection is closed, you could provide a 'finally' block in your code. A *finally* block always executes, regardless of an exception occurs or not.

To close the above opened connection, you should call close() method as follows –

```
conn.close();
```

Explicitly closing a connection conserves DBMS resources, which will make your database administrator happy.

. Using JDBC PreparedStatements

```
String prepState = "insert into albums values (?, ?);";

PreparedStatement prepState =
    connection.prepareStatement(sql);

prepState.setString(1, "Uprising");
prepState.setString(2, "Bob Marley and the Wailers ");

int rowsAffected = preparedStatement.executeUpdate();
```

```
//STEP 1. Import required packages
import java.sql.*;

public class FirstExample {
    // JDBC driver name and database URL
    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    static final String DB_URL = "jdbc:mysql://localhost/EMP";

    // Database credentials
    static final String USER = "username";
    static final String PASS = "password";

    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        try{
            //STEP 2: Register JDBC driver
            Class.forName("com.mysql.jdbc.Driver");

            //STEP 3: Open a connection
            System.out.println("Connecting to database...");
            conn = DriverManager.getConnection(DB_URL,USER,PASS);
```

```

//STEP 4: Execute a query
System.out.println("Creating statement...");
stmt = conn.createStatement();
String sql;
sql = "SELECT id, first, last, age FROM Employees";
ResultSet rs = stmt.executeQuery(sql);

//STEP 5: Extract data from result set
while(rs.next()){
    //Retrieve by column name
    int id   = rs.getInt("id");
    int age  = rs.getInt("age");
    String first = rs.getString("first");
    String last = rs.getString("last");

    //Display values
    System.out.print("ID: " + id);
    System.out.print(", Age: " + age);
    System.out.print(", First: " + first);
    System.out.println(", Last: " + last);
}

//STEP 6: Clean-up environment
rs.close();
stmt.close();
conn.close();

} catch (SQLException se) {
    //Handle errors for JDBC
    se.printStackTrace();
} catch (Exception e) {
    //Handle errors for Class.forName
    e.printStackTrace();
} finally{
    //finally block used to close resources
    try{
        if(stmt!=null)
            stmt.close();
    } catch(SQLException se2) {
        // nothing we can do
    }
    try{
        if(conn!=null)
            conn.close();
    } catch(SQLException se){
        se.printStackTrace();
    } //end finally try
} //end try
System.out.println("Goodbye!");

}//end main
}//end FirstExample

```

Now let us compile the above example as follows –

```
C:\>javac FirstExample.java  
C:\>
```

When you run **FirstExample**, it produces the following result –

```
C:\>java FirstExample  
Connecting to database...  
Creating statement...  
ID: 100, Age: 18, First: Zara, Last: Ali  
ID: 101, Age: 25, First: Mahnaz, Last: Fatma  
ID: 102, Age: 30, First: Zaid, Last: Khan  
ID: 103, Age: 28, First: Sumit, Last: Mittal  
C:\>
```

Once a connection is obtained we can interact with the database. The JDBC *Statement*, *CallableStatement*, and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

They also define methods that help bridge data type differences between Java and SQL data types used in a database.

The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

The Statement Objects

Creating Statement Object

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's `createStatement()` method, as in the following example –

```
Statement stmt = null;
try {
    stmt = conn.createStatement( );
    . . .
}
catch (SQLException e) {
    . . .
}
finally {
    . . .
}
```

Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

- **boolean execute (String SQL)**: Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate (String SQL)**: Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL)**: Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Closing Statement Object

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

```
Statement stmt = null;
try {
    stmt = conn.createStatement( );
    . . .
}
catch (SQLException e) {
    . . .
}
finally {
    stmt.close();
}
```

For a better understanding, we suggest you to study the [Statement - Example tutorial](#).

The PreparedStatement Objects

The *PreparedStatement* interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.

This statement gives you the flexibility of supplying arguments dynamically.

Creating PreparedStatement Object

```
String prepState = "insert into albums values (?, ?);";

PreparedStatement prepState =
    connection.prepareStatement(sql);

prepState.setString(1, "Uprising");
prepState.setString(2, "Bob Marley and the Wailers ");

int rowsAffected = preparedStatement.executeUpdate();
```

1. **import** java.sql.*;
2. **class** InsertPrepared{
3. **public static void** main(String args[]){
4. **try**{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
- 6.
7. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
- 8.
9. PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
10. stmt.setInt(1,101);//1 specifies the first parameter in the query
11. stmt.setString(2,"Ratan");
- 12.
13. **int** i=stmt.executeUpdate();
14. System.out.println(i+" records inserted");
- 15.
16. con.close();
- 17.
18. }**catch**(Exception e){ System.out.println(e);}

```
19.  
20.}  
21.}
```

All parameters in JDBC are represented by the ? symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

All of the **Statement object's** methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

Methods of PreparedStatement interface

The important methods of PreparedStatement interface are given below:

Method	Description
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)	sets the String value to the given parameter index.
public void setFloat(int paramIndex, float value)	sets the float value to the given parameter index.
public void setDouble(int paramIndex, double value)	sets the double value to the given parameter index.

public int executeUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

Closing PreparedStatement Object

Just as you close a Statement object, for the same reason you should also close the PreparedStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

```
PreparedStatement pstmt = null;
try {
    String SQL = "Update Employees SET age = ? WHERE id = ?";
    pstmt = conn.prepareStatement(SQL);
}
catch (SQLException e) {
    . . .
}
finally {
    pstmt.close();
}
```

For a better understanding, let us study [Prepare - Example Code](#).

The CallableStatement Objects

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

Creating CallableStatement Object

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$
CREATE PROCEDURE `EMP`.`getEmpName`
    (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
    SELECT first INTO EMP_FIRST
    FROM Employees
    WHERE ID = EMP_ID;
END $$
```

```
DELIMITER ;
```

Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.

Here are the definitions of each –

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

The following code snippet shows how to employ the **Connection.prepareCall()** method to instantiate a **CallableStatement** object based on the preceding stored procedure –

```
CallableStatement cstmt = null;
try {
    String SQL = "{call getEmpName (?, ?)}";
    cstmt = conn.prepareCall (SQL);
    . . .
}
catch (SQLException e) {
    . . .
}
finally {
    . . .
}
```

The String variable SQL, represents the stored procedure, with parameter placeholders.

Using the CallableStatement objects is much like using the PreparedStatement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQLException.

If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type, to the data type that the stored procedure is expected to return.

Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

Closing CallableStatement Object

Just as you close other Statement object, for the same reason you should also close the CallableStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the CallableStatement object as well. However, you should always explicitly close the CallableStatement object to ensure proper cleanup.

```
CallableStatement cstmt = null;
try {
    String SQL = "{call getEmpName (?, ?)}";
    cstmt = conn.prepareCall (SQL);
    . . .
}
catch (SQLException e) {
    . . .
}
finally {
    cstmt.close();
}
```

Statement Vs PreparedStatement Vs CallableStatement In Java :

Statement	PreparedStatement	CallableStatement
It is used to execute normal SQL queries.	It is used to execute parameterized or dynamic SQL queries.	It is used to call the stored procedures.

It is preferred when a particular SQL query is to be executed only once.	It is preferred when a particular query is to be executed multiple times.	It is preferred when the stored procedures are to be executed.
You cannot pass the parameters to SQL query using this interface.	You can pass the parameters to SQL query at run time using this interface.	You can pass 3 types of parameters using this interface. They are – IN, OUT and IN OUT.
This interface is mainly used for DDL statements like CREATE, ALTER, DROP etc.	It is used for any kind of SQL queries which are to be executed multiple times.	It is used to execute stored procedures and functions.
The performance of this interface is very low.	The performance of this interface is better than the Statement interface (when used for multiple execution of same query).	The performance of this interface is high.

Java ResultSetMetaData Interface

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

Commonly used methods of ResultSetMetaData interface

Method	Description

public int getColumnCount()throws SQLException	it returns the total number of columns in the ResultSet object.
public String getColumnName(int index)throws SQLException	it returns the column name of the specified column index.
public String getColumnTypeName(int index)throws SQLException	it returns the column type name for the specified index.
public String getTableName(int index)throws SQLException	it returns the table name for the specified column index.

How to get the object of ResultSetMetaData:

The getMetaData() method of ResultSet interface returns the object of ResultSetMetaData. Syntax:

1. **public** ResultSetMetaData getMetaData() **throws** SQLException
-

Example of ResultSetMetaData interface :

```

1. import java.sql.*;
2. class Rsmd{
3. public static void main(String args[]){
4. try{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6. Connection con=DriverManager.getConnection(
7. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
8.
9. PreparedStatement ps=con.prepareStatement("select * from emp");
10.ResultSet rs=ps.executeQuery();
11.ResultSetMetaData rsmd=rs.getMetaData();
12.
13.System.out.println("Total columns: "+rsmd.getColumnCount());
14.System.out.println("Column Name of 1st column: "+rsmd.getColumnName(1));
15.System.out.println("Column Type Name of 1st column: "+rsmd.getColumnTypeName(1));
16.
17.con.close();
18. }catch(Exception e){ System.out.println(e);}

```

```

19.}
20.}
    Output:Total columns: 2
        Column Name of 1st column: ID
        Column Type Name of 1st column: NUMBER

```

DatabaseMetaData Interface

The DatabaseMetaData is an interface that tells us the type of driver we are using, database product version, driver name, total number of table etc. It also provides all details about database providers.

DatabaseMetaData Interface Methods

Methods	Description
public Connection getConnection()	It retrieves the connection that produced the given metadata object.
public int getDatabaseMajorVersion()	Returns the major version number of the database.
public int getDatabaseMinorVersion()	Returns the minor version number of the database.
public String getDatabaseProductName()	Used to retrieve the name of this database product.
public String getDriverName()	Retrieves the name of the JDBC driver which is used in application.
public String getURL()	It returns the URL of the current DBMS.

Different types of ResultSet in JDBC.

The JDBC `java.sql.ResultSet` is used to handle the result returned from the SQL select statement.

The SQL select statements reads data from a database and return the data in a result set.

The result from a select statement is in a tabular form. It has columns and rows.

A ResultSet object maintains a cursor that points to the current row in the result set.

For a certain row we can use methods from `java.sql.ResultSet` to get the data column by column.

Based on the cursor scroll behavior, there are 3 types. The cursor is movable based on the properties of the ResultSet. These properties are set when creating the JDBC Statement.

ResultSet.TYPE_FORWARD_ONLY, the default type where cursor can only move forward in the result set.

ResultSet.TYPE_SCROLL_INSENSITIVE cursor can move forward and backward, and the resultset is insensitive to changes made by others to the database after the result set was created.

ResultSet.TYPE_SCROLL_SENSITIVE, the cursor can move In both direction, and the result set is sensitive to changes made by others to the database after the result set has been created.

Based on the concurrency there are 2 types of ResultSet object.

ResultSet.CONCUR_READ_ONLY: The result set is read only, this is the default concurrency type.

ResultSet.CONCUR_UPDATABLE: We can use ResultSet update method to update the row data.

Example

JDBC provides the following methods from Connection object to create statements with certain types of ResultSet.

- `createStatement(int resultSetType, int resultSetConcurrency);`
- `prepareStatement(String SQL, int resultSetType, int resultSetConcurrency);`
- `prepareCall(String sql, int resultSetType, int resultSetConcurrency);`

The following code creates a Statement object to create a forward-only, read only ResultSet object

```
Statement stmt = conn.createStatement()  
          ResultSet.TYPE_FORWARD_ONLY,
```

```
ResultSet.CONCUR_READ_ONLY);
```

Methods in ResultSet

The methods of the ResultSet interface have three categories:

- **Navigational methods** moves the cursor back and forth.
- **Getter methods** get the data from current row.
- **Update methods** update the data at the current row.

Commonly used methods of ResultSet interface

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.
6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.

8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.
11)beforeFirst()	Moves the cursor to before the first row
12)afterLast()	Moves the cursor to after the last row
13)int getRow()	Returns the row number that the cursor is pointing to.
14)moveToInsertRow()	Moves the cursor to where we can insert a new row into the database. The current row number is not changed.
15)moveToCurrentRow()	Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing

```

1. import java.sql.*;
2. class FetchRecord{
3.     public static void main(String args[])throws Exception{
4.
5.         Class.forName("oracle.jdbc.driver.OracleDriver");
6.         Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
7.         Statement stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
8.         ResultSet rs=stmt.executeQuery("select * from emp765");
9.

```

```

10. //getting the record of 3rd row
11. rs.absolute(3);
12. System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3));
13.
14. con.close();
15. }

```

ResultSet Navigation

We can use the following methods from ResultSet interface to move the cursor.

Methods	Description
beforeFirst()	Moves the cursor to before the first row
afterLast()	Moves the cursor to after the last row
first()	is used to move the cursor to the first row in result set object.
last()	Moves the cursor to the last row
absolute(int row)	Moves the cursor to the specified row
relative(int row)	Moves the cursor number of rows forward or backwards relative to where it is.
previous()	Moves the cursor to the previous row.
next()	Moves the cursor to the next row.
int getRow()	Returns the row number that the cursor is pointing to.
moveToInsertRow()	Moves the cursor to where we can insert a new row into the database. The current row number is not changed.

moveToCurrentRow()	Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing
--------------------	--

ResultSet Column Data

We have two ways to get data in the ResultSet.

- By Column Index
- By Column Name

For example, the following two methods get int value from a column. The first one is by column name and second one is by column index.

```
public int getInt(String columnName)  
public int getInt(int columnIndex)
```

The column index starts at 1.

Update ResultSet

We can update current row in ResultSet object.

We need to indicate the column name or index during the update.

For example, to update a String column of the current row we can use the the following methods.

```
public void updateString(int columnIndex, String s) throws SQLException  
public void updateString(String columnName, String s) throws SQLException
```

To push the update changes to the database, invoke one of the following methods.

Methods	Description
updateRow()	Updates the corresponding row in the database.
deleteRow()	Deletes the current row from the database

refreshRow()	Refreshes the result set to reflect any changes in the database.
cancelRowUpdates()	Cancels any updates made on the current row.
insertRow()	Inserts a row into the database when the cursor is pointing to the insert row.

Transaction Management in JDBC

Transaction represents **a single unit of work**.

The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.

Atomicity means either all successful or none.

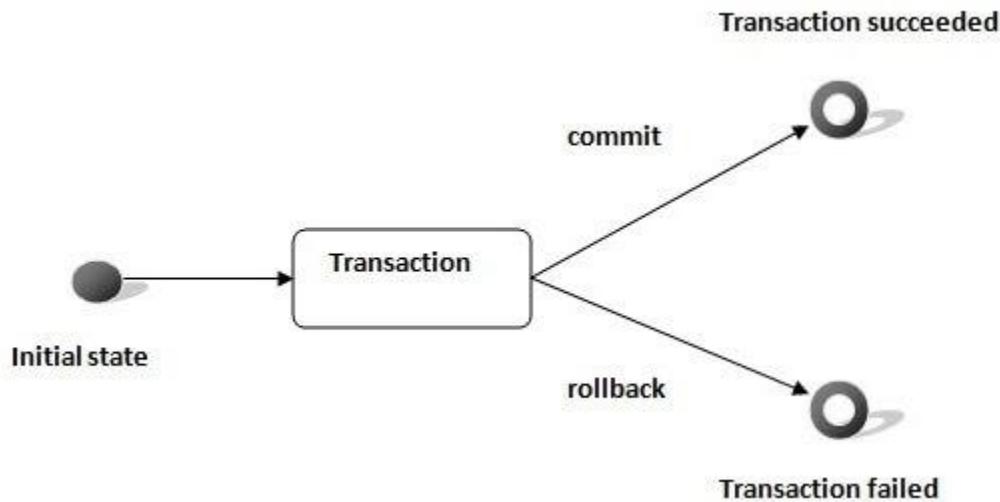
Consistency ensures bringing the database from one consistent state to another consistent state.

Isolation ensures that transaction is isolated from other transaction.

Durability means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

Advantage of Transaction Management

fast performance It makes the performance fast because database is hit at the time of commit.



In JDBC, **Connection interface** provides methods to manage transaction.

Method	Description
<code>void setAutoCommit(boolean status)</code>	It is true by default means each transaction is committed by default.
<code>void commit()</code>	commits the transaction.
<code>void rollback()</code>	cancels the transaction.

Atomicity

By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

—**Abort**: If a transaction aborts, changes made to database are not visible.

—**Commit**: If a transaction commits, changes made are visible.

Atomicity is also known as the 'All or nothing rule'.

Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.

Before: X : 500	Y: 200
Transaction T	
T1	T2
Read (X) X: = X - 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of **T1** but before completion of **T2**.(say, after **write(X)** but before **write(Y)**), then amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

Consistency

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above,

The total amount before and after the transaction must be maintained.

Total **before T occurs** = **500 + 200 = 700**.

Total **after T occurs** = **400 + 300 = 700**.

Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result T is incomplete.

Isolation

This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

Let **X= 500, Y = 500**.

Consider two transactions **T** and **T''**.

T	T''
Read (X)	Read (X)
X: = X*100	Read (Y)
Write (X)	Z: = X + Y
Read (Y)	Write (Z)
Y: = Y - 50	
Write	

Suppose **T** has been executed till **Read (Y)** and then **T''** starts. As a result , interleaving of operations takes place due to which **T''** reads correct value of **X** but incorrect value of **Y** and sum computed by

$$T'': (X+Y = 50,000 + 500 = 50,500)$$

is thus not consistent with the sum at end of transaction:

$$T: (X+Y = 50,000 + 450 = 50,450).$$

This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

Durability:

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

The **ACID** properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored

Simple example of transaction management in jdbc using Statement

Let's see the simple example of transaction management using Statement.

```
1. import java.sql.*;
2. class FetchRecords{
3. public static void main(String args[])throws Exception{
4. Class.forName("oracle.jdbc.driver.OracleDriver");
5. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
6. con.setAutoCommit(false);
7.
8. Statement stmt=con.createStatement();
9. stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
10. stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");
11.
12.con.commit();
13.con.close();
14.}}
```

If you see the table emp400, you will see that 2 records has been added.

Example of transaction management in jdbc using PreparedStatement

Let's see the simple example of transaction management using PreparedStatement.

```
1. import java.sql.*;
2. import java.io.*;
3. class TM{
4. public static void main(String args[]){
5. try{
6.
7. Class.forName("oracle.jdbc.driver.OracleDriver");
8. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
9. con.setAutoCommit(false);
10.
11.PreparedStatement ps=con.prepareStatement("insert into user420 values(?, ?, ?)");
12.
13.BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
14.while(true){
15.
16.System.out.println("enter id");
17.String s1=br.readLine();
18.int id=Integer.parseInt(s1);
19.
20.System.out.println("enter name");
21.String name=br.readLine();
22.
23.System.out.println("enter salary");
24.String s3=br.readLine();
25.int salary=Integer.parseInt(s3);
26.
27.ps.setInt(1,id);
28.ps.setString(2,name);
29.ps.setInt(3,salary);
30.ps.executeUpdate();
31.
32.System.out.println("commit/rollback");
33.String answer=br.readLine();
34.if(answer.equals("commit")){
35.
36.ps.commit();
37.ps.close();
38.con.close();
39.
40.out.println("Data Inserted");
41.
42.out.println("Program Ended");
43.
44.
45.
46.
47.
48.
49.
50.
51.
52.
53.
54.
55.
56.
57.
58.
59.
59.
60.
61.
62.
63.
64.
65.
66.
67.
68.
69.
69.
70.
71.
72.
73.
74.
75.
76.
77.
78.
79.
79.
80.
81.
82.
83.
84.
85.
86.
87.
88.
89.
89.
90.
91.
92.
93.
94.
95.
96.
97.
98.
99.
99.
```

```
35.con.commit();
36.}
37.if(answer.equals("rollback")){
38.con.rollback();
39.}
40.
41.
42.System.out.println("Want to add more records y/n");
43.String ans=br.readLine();
44.if(ans.equals("n")){
45.break;
46.}
47.
48.}
49.con.commit();
50.System.out.println("record successfully saved");
51.
52.con.close();//before closing connection commit() is called
53.}catch(Exception e){System.out.println(e);}
54.
55.}}
```

It will ask to add more records until you press n. If you press n, transaction is committed.

Batch Processing in JDBC

Instead of executing a single query, we can execute a batch (group) of queries. It makes the performance fast.

The `java.sql.Statement` and `java.sql.PreparedStatement` interfaces provide methods for batch processing.

Advantage of Batch Processing

Fast Performance

Methods of Statement interface

The required methods for batch processing are given below:

Method	Description
<code>void addBatch(String query)</code>	It adds query into batch.
<code>int[] executeBatch()</code>	It executes the batch of queries.

- The **addBatch()** method of `Statement`, `PreparedStatement`, and `CallableStatement` is used to add individual statements to the batch. The **executeBatch()** is used to start the execution of all the statements grouped together.
- The **executeBatch()** returns an array of integers, and each element of the array represents the update count for the respective update statement.
- Just as you can add statements to a batch for processing, you can remove them with the **clearBatch()** method. This method removes all the statements you added with the `addBatch()` method. However, you cannot selectively choose which statement to remove.

Example of batch processing in jdbc

Let's see the simple example of batch processing in jdbc. It follows following steps:

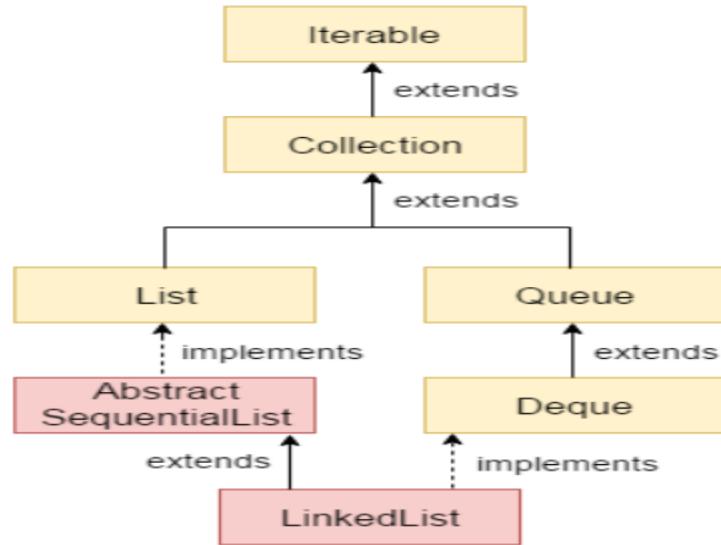
- o Load the driver class
- o Create Connection
- o Create Statement
- o Add query in the batch
- o Execute Batch
- o Close Connection

```
1. import java.sql.*;  
2. class FetchRecords{  
3. public static void main(String args[])throws Exception{  
4. Class.forName("oracle.jdbc.driver.OracleDriver");  
5. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");  
5. con.setAutoCommit(false);  
7.  
3. Statement stmt=con.createStatement();  
9. stmt.addBatch("insert into user420 values(190,'abhi',40000)");  
10. stmt.addBatch("insert into user420 values(191,'umesh',50000)");  
11.  
12. stmt.executeBatch();//executing the batch  
13.  
14. con.commit();  
15. con.close();  
16. }}}
```

LINKED LIST IN JAVA

- Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure.
- It inherits the AbstractList class and implements List and Deque interfaces.
- In linked list, elements are not stored in contiguous locations like arrays, they are linked with each other using pointers.
- Linked list class maintains insertion order.
- Java LinkedList class is non synchronized, it is not thread safe.
- Java LinkedList class can contain duplicate elements.

- Java LinkedList class can be used as a list, stack or queue.



- The Java LinkedList class provide double linked list implementation.

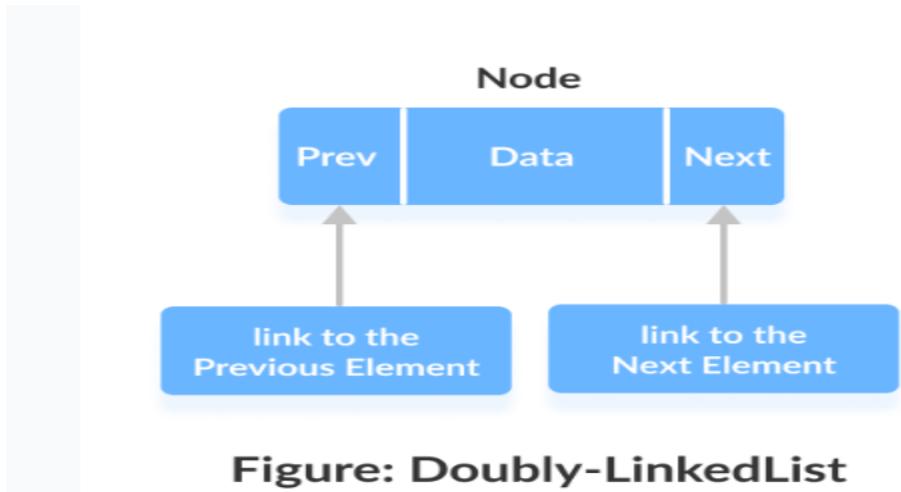


Figure: Doubly-LinkedList

Each element in a linked list is known as a **node**. It consists of 3 fields:

- Prev** - Stores an address of the previous element in the list. It is null for the first element.
- Next** - Stores an address of the next element in the list. It is null for the last element.
- Data** - Stores the actual data.

Constructors of Java LinkedList

Constructor	Description
LinkedList()	It is used to construct an empty list.
LinkedList(Collection<? extends E> c)	It is used to construct a list containing the elements of the specified collection, in the order, they are returned by the collection's iterator.

Creating a LinkedList

- Here is how we can create linkedlist in java

```
LinkedList<Type> linkedList = new LinkedList<>();
```

Here, type indicates the type of linked list. For example

```
// create Integer type linked list
LinkedList<Integer> linkedList = new LinkedList<>();

// create String type linked list
LinkedList<String> linkedList = new LinkedList<>();
```

- We can create linkedlist using interfaces.

```
List<String> animals1 = new LinkedList<>();
```

Here we have declared a linked list, `animals1`, using the `List` interface. The linked list can only access the methods of the `List` interface.

```
Queue<String> animals2 = new LinkedList<>();
Deque<String> animals3 = new LinkedList<>();
```

Here, `animals2` can only access the methods of the `Queue` interface.

However, `animals3` can only access the methods of the `Deque` and `Queue` interfaces. It's because `Deque` is a subinterface of `Queue`.

Methods of LinkedList

- Linkedlist allows different methods to perform various actions.

boolean add(Object item): It adds the item at the end of the list.

void add(int index, Object item): It adds an item at the given index of the the list.

boolean addAll(Collection c): It adds all the elements of the specified collection c to the list.

boolean addAll(int index, Collection c): It adds all the elements of collection c to the list starting from a give index in the list.

void addFirst(Object item): It adds the item (or element) at the first position in the list.

void addLast(Object item): It inserts the specified item at the end of the list.

offer(E e): This method Adds the specified element as the tail (last element) of this list.

offerFirst(E e): This method Inserts the specified element at the front of this list.

offerLast(E e): This method Inserts the specified element at the end of this list.

void clear(): It removes all the elements of a list.

Object clone(): It returns the copy of the list.

We can also use the **listsIterator()** method to add elements to the linked list.

The **difference** is that **offer()** will return false if it fails to insert the element on a size restricted Queue, whereas **add()** will throw an IllegalStateException .

Example:

```
import java.util.*;
```

```
class Main {  
    public static void main(String[] args){  
        LinkedList<String> animals = new LinkedList<>();  
    }  
}
```

```
// Add elements to LinkedList  
animals.add("Dog");  
animals.add("Cat");  
animals.add("Horse");  
System.out.println("LinkedList: " + animals);  
  
//adding element at specified Index  
animals.add(1,"Rabbit");  
System.out.println("LinkedList: " + animals);  
  
//adding specified collection to the list  
LinkedList<String> birds= new LinkedList<>();  
birds.add("pigeon");  
animals.addAll(birds);  
System.out.println("LinkedList: " + animals);  
  
//adding the elements at specified Index  
animals.addAll(2,birds);  
System.out.println("LinkedList: " + animals);  
  
//adding elements using listiterator  
ListIterator<String> listiterate = animals.listIterator();  
listiterate.add("cow");  
System.out.println("LinkedList: " + animals);  
  
//adding the element at the first Index  
animals.addFirst("fox");
```

```

System.out.println("LinkedList: " + animals);

//adding the element at the last Index
animals.addLast("peacock");
System.out.println("LinkedList: " + animals);

//copying of list
Object list=animals.clone();
System.out.println("LinkedList: " + list);

//removing all elements
animals.clear();
System.out.println("LinkedList: " + animals);

}

}

```

Output:

```

LinkedList: [Dog, Cat, Horse]
LinkedList: [Dog, Rabbit, Cat, Horse]
LinkedList: [Dog, Rabbit, Cat, Horse, pigeon]
LinkedList: [Dog, Rabbit, pigeon, Cat, Horse, pigeon]
LinkedList: [cow, Dog, Rabbit, pigeon, Cat, Horse, pigeon]
LinkedList: [fox, cow, Dog, Rabbit, pigeon, Cat, Horse, pigeon]
LinkedList: [fox, cow, Dog, Rabbit, pigeon, Cat, Horse, pigeon, peacock]
LinkedList: [fox, cow, Dog, Rabbit, pigeon, Cat, Horse, pigeon, peacock]
LinkedList: []

```

boolean contains(Object item): It checks whether the given item is present in the list or not. If the item is present then it returns true else false.

Object get(int index): It returns the item of the specified index from the list.

Object getFirst(): It fetches the first item from the list.

Object getLast(): It fetches the last item from the list.

int indexOf(Object item): It returns the index of the specified item. returns -1, if element not present in the list.

Object poll(): It returns and removes the first item of the list.

Object pollFirst(): same as poll() method. Removes the first item of the list.

Object pollLast(): It returns and removes the last element of the list.

peekFirst(): Retrieves, but does not remove, the first element of this list, or returns null if this list is empty.

peekLast(): Retrieves, but does not remove, the last element of this list, or returns null if this list is empty.

Pop(): This method Pops an element from the stack represented by this list.

Push(E e): This method Pushes an element onto the stack represented by this list.

Object remove(): It removes the first element of the list.

Object remove(int index): It removes the item from the list which is present at the specified index.

Object remove(Object obj): It removes the specified object from the list.

Object removeFirst(): It removes the first item from the list.

Object removeLast(): It removes the last item of the list.

Object removeFirstOccurrence(Object item): It removes the first occurrence of the specified item and returns true otherwise returns false.

Object removeLastOccurrence(Object item): It removes the last occurrence of the given element.

Object set(int index, Object item): It updates the item of specified index with the give value.

int size(): It returns the number of elements of the list.

Example:

```
import java.util.*;
```

```
public class Main {  
    public static void main(String[] args){  
        LinkedList<String> animals = new LinkedList<>();  
  
        // Add elements to LinkedList  
        animals.add("Dog");  
        animals.add("Cat");  
        animals.add("Horse");  
        animals.add("giraffe");  
        animals.add("lion");  
        animals.add("tiger");  
        animals.add("bear");  
        animals.add("Doggy");  
        animals.add("Cat");  
        System.out.println("LinkedList: " + animals);  
  
        //checking if elemet present  
        if(animals.contains("Cat"))  
            System.out.println("exixts");  
        else  
            System.out.println("not exixts");  
  
        //returning item of specified Index  
        System.out.println(animals.get(1));  
  
        //returning first elemet from the list  
        System.out.println(animals.getFirst());
```

```
//returning last elemet from the list
System.out.println(animals.getLast());

//returning index of specified item
System.out.println(animals.indexOf("eref"));
System.out.println(animals.indexOf("Dog"));

System.out.println("LinkedList: " + animals);

//removing elements from the list
System.out.println(animals.remove());
System.out.println(animals.remove(2));
System.out.println(animals.remove("Dog"));
System.out.println(animals.removeFirst());
System.out.println(animals.removeLast());
System.out.println(animals.removeFirstOccurrence("Cat"));
System.out.println(animals.removeLastOccurrence("bear"));
System.out.println("LinkedList: "+animals);

//peek methods
System.out.println(animals.peekFirst());
System.out.println(animals.peekLast());
System.out.println("LinkedList: "+animals);

//setting an elements in the list
System.out.println(animals.set(1,"cat"));
System.out.println("LinkedList: "+animals);
```

```
//size of the list  
System.out.println(animals.size());  
  
}  
}
```

Output:

LinkedList: [Dog, Cat, Horse, giraffe, lion, tiger, bear, Doggy, Cat]

exists

Cat

Dog

Cat

-1

0

LinkedList: [Dog, Cat, Horse, giraffe, lion, tiger, bear, Doggy, Cat]

Dog

giraffe

false

Cat

Cat

false

true

LinkedList: [Horse, lion, tiger, Doggy]

Horse

Doggy

LinkedList: [Horse, lion, tiger, Doggy]

lion

LinkedList: [Horse, cat, tiger, Doggy]

4

Note:

- Here, both poll and remove are used to delete elements in the linked list.
- But, the difference to be noted is when the queue is empty
 Poll returns null
 Whereas remove throws NoSuchElementException.

Execution:

```
import java.util.*;  
  
public class Main  
{  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
  
        Queue<String> days = new LinkedList<>();  
        System.out.println("Queue: "+days);  
  
        //poll  
        System.out.println(days.poll());  
  
        //remove  
        System.out.println(days.remove());  
  
    }  
}
```

Output:

```
Hello World
Queue: []
null
Exception in thread "main" java.util.NoSuchElementException
    at java.util.LinkedList.removeFirst(LinkedList.java:270)
    at java.util.LinkedList.remove(LinkedList.java:685)
    at Main.main(Main.java:15)
```

We can access the elements of list using iterator() and listIterator() methods also.

Access Elements: using iterator() method:

To iterate over the elements of a linked list, we can use the `iterator()` method. We must import `java.util.Iterator` package to use this method. For example,

```
import java.util.LinkedList;
import java.util.Iterator;

class Main {
    public static void main(String[] args) {
        LinkedList<String> animals= new LinkedList<>();

        // Add elements in LinkedList
        animals.add("Dog");
        animals.add("Horse");
        animals.add("Cat");

        // Creating an object of Iterator
        Iterator<String> iterate = animals.iterator();
        System.out.print("LinkedList: ");
```

```
        while(iterate.hasNext()) {  
            System.out.print(iterate.next());  
            System.out.print(", ");  
        }  
    }  
}
```

Output

LinkedList: Dog, Cat, Horse,

Here,

hasNext()→returns true if there is a next element.

next()→returns the next element.

Access Elements: using the listIterator() Method:

We can also use the `listIterator()` method to iterate over the elements of a linked list. To use this method, we must import `java.util.ListIterator` package.

The `listIterator()` method is more preferred in linked lists. It is because objects of `listIterator()` can iterate backward as well. For example,

```
import java.util.LinkedList;  
import java.util.ListIterator;  
  
class Main {  
    public static void main(String[] args) {
```

```
LinkedList<String> animals= new LinkedList<>();  
  
// Add elements in LinkedList  
animals.add("Dog");  
animals.add("Horse");  
animals.add("Cat");  
  
// Create an object of ListIterator  
ListIterator<String> listIterate = animals.listIterator();  
System.out.print("LinkedList: ");  
  
while(listIterate.hasNext()) {  
    System.out.print(listIterate.next());  
    System.out.print(", ");  
}  
  
// Iterate backward  
System.out.print("\nReverse LinkedList: ");  
  
while(listIterate.hasPrevious()) {  
    System.out.print(listIterate.previous());  
    System.out.print(", ");  
}  
}  
}
```

Output:

LinkedList: Dog, Horse, Cat,
Reverse LinkedList: Cat, Horse, Dog,

Here,

`hasNext()`→returns true if there is a next element.

`next()`→returns the next element.

`hasPrevious()`→returns true if there is any previous element.

`previous()`→returns the previous element.

Advantages of Linked List

- **Dynamic Data Structure**
- **Insertion and deletion**
- No memory wastage

Disadvantages Of linked list:

- Memory usage
- Traversal

REFERENCES:

<https://www.geeksforgeeks.org/linked-list-in-java/>

<https://beginnersbook.com/2013/12/linkedlist-in-java-with-example/>

<https://www.programiz.com/java-programming/linkedlist>

<https://www.javatpoint.com/java-linkedlist>

LinkedHashMap in Java

LinkedHashMap is just like [HashMap](#) with an additional feature of maintaining an order of elements inserted into it. HashMap provided the advantage of quick insertion, search and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order. Few important features of LinkedHashMap are as follows:

- A LinkedHashMap contains values based on the key. It implements the Map interface and extends HashMap class.
- It contains only unique elements ..
- It may have one null key and multiple null values .
- It is same as HashMap with additional feature that it maintains insertion order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

Constructors in LinkedHashMap:

The LinkedHashMap accepts five types of constructors:

1. **LinkedHashMap():** This is used to construct a default LinkedHashMap constructor.
2. **LinkedHashMap(int capacity):** It is used to initialize a particular LinkedHashMap with a specified capacity.
3. **LinkedHashMap(Map m_a_p):** It is used to initialize a particular LinkedHashMap with the elements of the specified map.
4. **LinkedHashMap(int capacity, float fillRatio):** It is used to initialize both the capacity and fill ratio for a LinkedHashMap.
5. **LinkedHashMap(int capacity, float fillRatio, boolean Order):** This constructor is also used to initialize both the capacity and fill ratio for a LinkedHashMap along with whether to follow the insertion order or not.
 - True is passed for last access order.
 - False is passed for insertion order.

Methods in LinkedHashMap:

1. **void clear():** This method is used to remove all the mappings from the map.
2. **boolean containsKey(Object key):** This method is used to returns true if a specified element is mapped by one or more keys.
3. **Object get(Object key):** The method is used to retrieve or fetch the value mapped by the specified key.
4. **protected boolean removeEldestEntry(Map.Entry eldest):** The method is used to return true when the map removes its eldest entry from the map.
5. **entrySet?():** This method returns a Set view of the mappings contained in this map.
6. **forEach?(BiConsumer<K,V> action):** This method Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
7. **getOrDefault?(Object key, V defaultValue):** This method returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
8. **keySet?():** This method returns a Set view of the keys contained in this map.
9. **removeEldestEntry?(Map.Entry<K,V> eldest):** This method returns true if this map should remove its eldest entry.

10. **replaceAll?(BiFunction<K,V> function)**: This method replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

11. **values?()**: This method returns a Collection view of the values contained in this map.

REFERENCE LINK:

<https://www.geeksforgeeks.org/linkedhashmap-class-java-examples/>

LinkedHashSet in Java

A LinkedHashSet is an ordered version of [HashSet](#) that maintains a doubly-linked List across all elements. When the iteration order is needed to be maintained this class is used. When iterating through a [HashSet](#) the order is unpredictable, while a LinkedHashSet lets us iterate through the elements in the order in which they were inserted. When cycling through LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.

Syntax:

```
LinkedHashSet<String> hs = new LinkedHashSet<String>();
```

- Contains unique elements only like [HashSet](#). It extends [HashSet](#) class and implements Set interface.
- Maintains insertion order.

HashSet uses **HashMap object** internally to store it's elements whereas LinkedHashSet uses **LinkedHashMap object** internally to store and process it's elements. In this article, we will see how LinkedHashSet works internally and how it maintains insertion order.

Given below are the list of constructors supported by the LinkedHashSet:

1. **LinkedHashSet()**: This constructor is used to create a default HashSet.
2. **LinkedHashSet(Collection C)**: Used in initializing the HashSet with the elements of the collection C
3. **LinkedHashSet(int size)**: Used to initialize the size of the LinkedHashSet with the integer mentioned in the parameter.
4. **LinkedHashSet(int capacity, float fillRatio)**: Can be used to initialize both the capacity and the fill ratio, also called the load capacity of the LinkedHashSet with the arguments mentioned in the parameter. When the number of elements exceeds the capacity of the hash set is multiplied with the fill ratio thus expanding the capacity of the LinkedHashSet.

Here, we have created a linked hash set named `numbers`.

Notice, the part `new LinkedHashSet<>(8, 0.75)`. Here, the first parameter is **capacity** and the second parameter is **loadFactor**.

- **capacity** - The capacity of this hash set is 8. Meaning, it can store 8 elements.
- **loadFactor** - The load factor of this hash set is 0.6. This means, whenever our hash table is filled by 60%, the elements are moved to a new hash table of double the size of the original hash table.

Default capacity and load factor

It's possible to create a linked hash set without defining its capacity and load factor. For example,

```
// LinkedHashSet with default capacity and load factor  
LinkedHashSet<Integer> numbers1 = new LinkedHashSet<>();
```

By default,

- the capacity of the linked hash set will be 16
- the load factor will be 0.75

LinkedHashSet doesn't have its own methods. All methods are inherited from its super class i.e HashSet. So, all operations on LinkedHashSet work in the same manner as that of HashSet. The only change is the internal object used to store the elements. In HashSet, elements you insert are stored as **keys of HashMap** object. Whereas in LinkedHashSet, elements you insert are stored as **keys of LinkedHashMap** object. The values of these keys will be the same constant i.e "PRESENT".

```
1. import java.util.*;  
2. class LinkedHashSet1{  
3. public static void main(String args[]){  
4. //Creating HashSet and adding elements  
5.     LinkedHashSet<String> set=new LinkedHashSet();  
6.         set.add("One");  
7.         set.add("Two");  
8.         set.add("Three");  
9.         set.add("Four");  
10.            set.add("Five");
```

```
11.         Iterator<String> i=set.iterator();
12.         while(i.hasNext())
13.         {
14.             System.out.println(i.next());
15.         }
16.     }
17. }
```

Output:

```
One
Two
Three
Four
Five
```

Java LinkedHashSet example ignoring duplicate Elements

```
1. import java.util.*;
2. class LinkedHashSet2{
3.     public static void main(String args[]){
4.         LinkedHashSet<String> al=new LinkedHashSet<String>();
5.         al.add("Ravi");
6.         al.add("Vijay");
7.         al.add("Ravi");
8.         al.add("Ajay");
9.         Iterator<String> itr=al.iterator();
10.        while(itr.hasNext()){
11.            System.out.println(itr.next());
12.        }
13.    }
14. }
```

Output:

```
Ravi
Vijay
Ajay
```

How LinkedHashSet Maintains Insertion Order?

LinkedHashSet uses LinkedHashMap object to store it's elements. The elements you insert in the LinkedHashSet are stored as keys of this LinkedHashMap object. Each **key, value pair** in the LinkedHashMap are instances of it's static inner class called **Entry<K, V>**. This Entry<K, V> class extends **HashMap.Entry** class. The insertion order of elements into LinkedHashMap are maintained by adding two new fields to this class. They are **before** and **after**. These two fields hold the references to previous and next elements. These two fields make LinkedHashMap to function as a doubly linked list.

```
1  private static class Entry<K,V> extends HashMap.Entry<K,V>
2  {
3      // These fields comprise the doubly linked list used for iteration.
4      Entry<K,V> before, after;
5      Entry(int hash, K key, V value, HashMap.Entry<K,V> next) {
6          super(hash, key, value, next);
7      }
8  }
```

The first two fields of above inner class of LinkedHashMap – **before** and **after** are responsible for maintaining the insertion order of the LinkedHashSet. The header field of LinkedHashMap stores the head of this doubly linked list. It is declared like below,

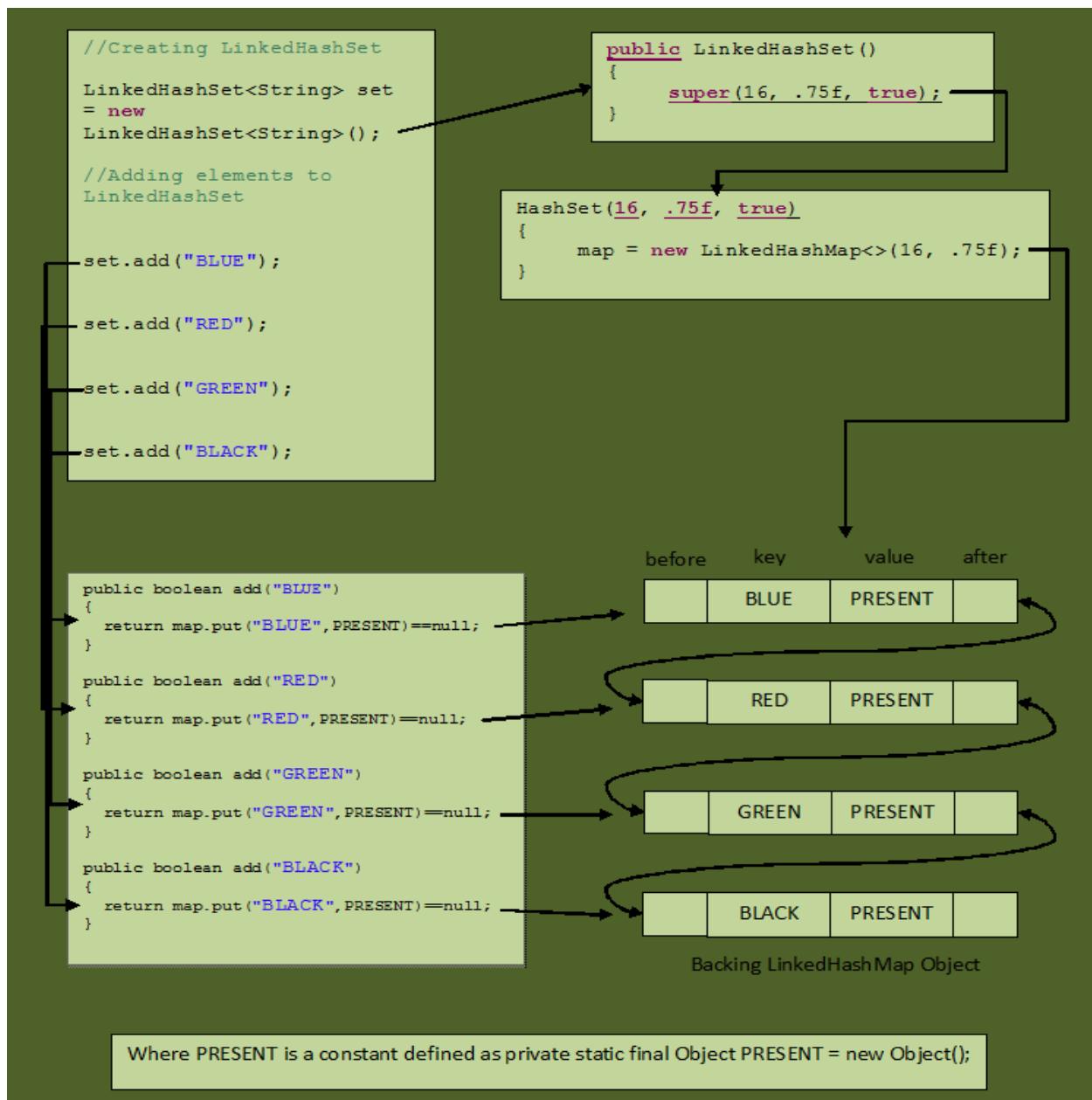
```
1  private transient Entry<K,V> header;      //Stores the head of the doubly linked list
```

In LinkedHashMap, the same set of Entry objects (rather references to Entry objects) are arranged in two different manner. One is the HashMap and another one is Doubly linked list. The Entry objects just sit on heap memory, unaware of that they are part of two different data structures.

Let's see one example of LinkedHashSet to know how it works internally.

```
1  public class LinkedHashSetExample
2  {
```

```
3     public static void main(String[] args)
4     {
5         //Creating LinkedHashSet
6
7         LinkedHashSet<String> set = new LinkedHashSet<String>();
8
9         //Adding elements to LinkedHashSet
10
11        set.add("BLUE");
12
13        set.add("RED");
14
15        set.add("GREEN");
16
17        set.add("BLACK");
18    }
19 }
```



This **boolean dummy value** is just used to differentiate this constructor from other constructors of `HashSet` class which take initial capacity and load factor as their arguments. Here is the how this constructor is defined in `HashSet` class.

```
1  HashSet(int initialCapacity, float loadFactor, boolean dummy)
2  {
3      map = new LinkedHashMap<>(initialCapacity, loadFactor);
```

REFERENCE LINK:

<https://javaconceptoftheday.com/how-linkedhashset-works-internally-in-java/>

LinkedHashSet Methods

1. **public boolean add(E e)** : adds the specified element to the Set if not already present. This method internally uses **equals()** method to check for duplicates. If element is duplicate then element is rejected and value is NOT replaced.
2. **public void clear()** : removes all the elements from the LinkedHashSet.
3. **public boolean contains(Object o)** : returns true if the LinkedHashSet contains the specified element, otherwise false.
4. **public boolean isEmpty()** : returns true if LinkedHashSet contains no element, otherwise false.
5. **public int size()** : returns the number of elements in the LinkedHashSet.
6. **public Iterator<E> iterator()** : returns an iterator over the elements in this LinkedHashSet. The elements are returned from iterator in no specific order.
7. **public boolean remove(Object o)** : removes the specified element from the LinkedHashSet if it is present and return true, else returns false.
8. **public boolean removeAll(Collection<?> c)** : remove all the elements in the LinkedHashSet that are part of the specified collection.
9. **public Object clone()** : returns a shallow copy of the LinkedHashSet.
10. **public Spliterator<E> spliterator()** : creates a late-binding and fail-fast Spliterator over the elements in this LinkedHashSet. It has following initialization properties **Spliterator.DISTINCT**, **Spliterator.ORDERED**.

LinkedHashSet add, remove, iterator example

Java LinkedHashSet Example

//1. Create LinkedHashSet

```
LinkedHashSet<String> LinkedHashSet = new LinkedHashSet<>();
```

```
//2. Add elements to LinkedHashSet  
LinkedHashSet.add("A");  
LinkedHashSet.add("B");  
LinkedHashSet.add("C");  
LinkedHashSet.add("D");  
LinkedHashSet.add("E");  
System.out.println(LinkedHashSet);  
  
//3. Check if element exists  
boolean found = LinkedHashSet.contains("A");      //true  
System.out.println(found);  
  
//4. Remove an element  
LinkedHashSet.remove("D");  
  
//5. Iterate over values  
Iterator<String> itr = LinkedHashSet.iterator();  
while(itr.hasNext())  
{  
    String value = itr.next();  
    System.out.println("Value: " + value);  
}
```

Program Output.

```
[A, B, C, D, E]  
true  
Value: A  
Value: B  
Value: C  
Value: E
```

5.2. Convert LinkedHashSet to Array Example

Java example to convert a LinkedHashSet to array using `toArrray()` method.

```
//Java LinkedHashSet Example
```

```
LinkedHashSet<String> LinkedHashSet = new LinkedHashSet<>();  
LinkedHashSet.add("A");  
LinkedHashSet.add("B");  
LinkedHashSet.add("C");  
LinkedHashSet.add("D");  
LinkedHashSet.add("E");  
String[] values = new String[LinkedHashSet.size()];  
LinkedHashSet.toArray(values);  
System.out.println(Arrays.toString(values));
```

Program Output.

```
[A, B, C, D, E]
```

Convert LinkedHashSet to ArrayList Example

Java example to convert a LinkedHashSet to arraylist using [Java 8 stream API](#).

```
//Java LinkedHashSet Example
```

```
LinkedHashSet<String> LinkedHashSet = new LinkedHashSet<>();  
LinkedHashSet.add("A");  
LinkedHashSet.add("B");  
LinkedHashSet.add("C");  
LinkedHashSet.add("D");  
LinkedHashSet.add("E");  
List<String> valuesList = LinkedHashSet.stream().collect(Collectors.toList());  
System.out.println(valuesList);
```

Program Output.

[A, B, C, D, E]

HashSet vs. TreeSet vs. LinkedHashSet

HashSet is Implemented using a hash table. Elements are not ordered.

The add, remove, and contains methods have constant time complexity O(1).

TreeSet is implemented using a tree structure(red-black tree in algorithm book). The elements in a set are sorted, but the add, remove, and contains methods has time complexity of O(log (n)). It offers several methods to deal with the ordered set like first(), last(), headSet(), tailSet(), etc.

LinkedHashSet is between HashSet and TreeSet. It is implemented as a hash table with a linked list running through it, so it provides the order of insertion. The time complexity of basic methods is O(1).

LinkedHashSet Vs. HashSet

Both `LinkedHashSet` and `HashSet` implements the `Set` interface. However, there exist some differences between them.

- `LinkedHashSet` maintains a linked list internally. Due to this, it maintains the insertion order of its elements.
- The `LinkedHashSet` class requires more storage than `HashSet`. This is because `LinkedHashSet` maintains linked lists internally.
- The performance of `LinkedHashSet` is slower than `HashSet`. It is because of linked lists present in `LinkedHashSet`.

LinkedHashSet Vs. TreeSet

Here are the major differences between `LinkedHashSet` and `TreeSet`:

- The `TreeSet` class implements the `SortedSet` interface. That's why elements in a tree set are sorted. However, the `LinkedHashSet` class only maintains the insertion order of its elements.

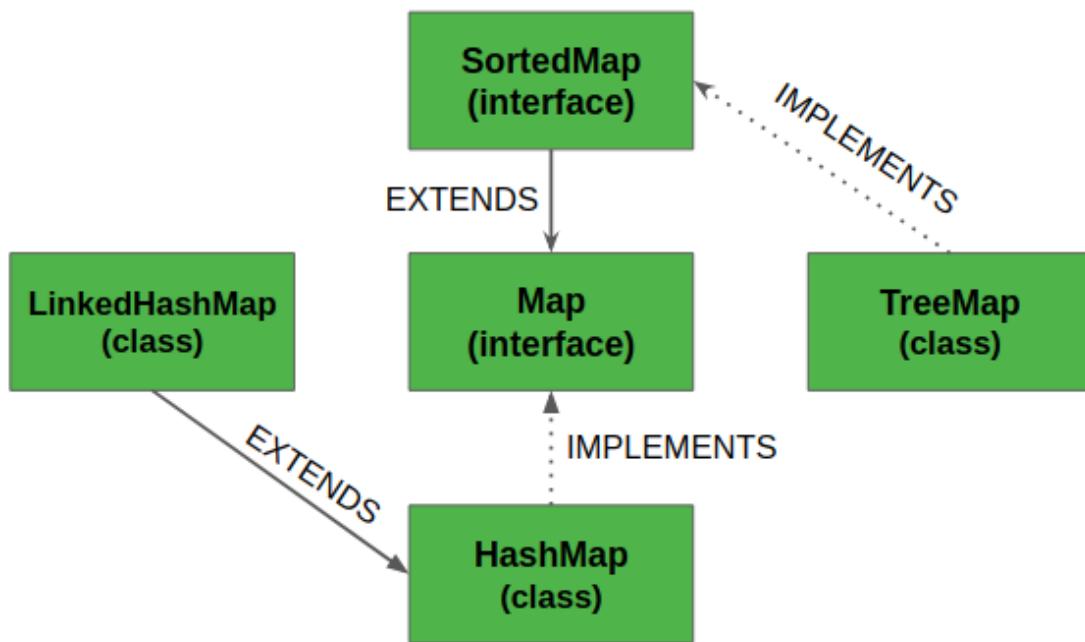
- A `TreeSet` is usually slower than a `LinkedHashSet`. It is because whenever an element is added to a `TreeSet`, it has to perform the sorting operation.
- `LinkedHashSet` allows the insertion of null values. However, we cannot insert a null value to `TreeSet`.

Map interface in java

The `java.util.Map` interface represents a mapping between a key and a value. The `Map` interface is not a subtype of the `Collection` interface. Therefore it behaves a bit different from the rest of the collection types.

Few characteristics of the Map Interface are:

1. A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value like the `HashMap` and `LinkedHashMap`, but some do not like the `TreeMap`.
2. The order of a map depends on specific implementations, e.g `TreeMap` and `LinkedHashMap` have predictable order, while `HashMap` does not.
3. There are two interfaces for implementing Map in java: `Map` and `SortedMap`, and three classes: `HashMap`, `TreeMap` and `LinkedHashMap`.



MAP Hierarchy in Java

Why and When to use Maps?

Maps are perfect to use for key-value association mapping such as dictionaries. The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys. Some examples are:

- A map of error codes and their descriptions.
- A map of zip codes and cities.
- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).

Methods in Map Interface:

1. `public Object put(Object key, Object value)`: This method is used to insert an entry in this map.
2. `public void putAll(Map map)`: This method is used to insert the specified map in this map.
3. `public Object remove(Object key)`: This method is used to delete an entry for the specified key.
4. `public Object get(Object key)`: This method is used to return the value for the specified key.
5. `public boolean containsKey(Object key)`: This method is used to search the specified key from this map.

6. public Set keySet(): This method is used to return the Set view containing all the keys.
7. public Set entrySet(): This method is used to return the Set view containing all the keys and values.

```
// Java program to demonstrate working of Map interface

import java.util.*;
class HashMapDemo
{
    public static void main(String args[])
    {
        Map< String, Integer> hm =
            new HashMap< String, Integer>();
        hm.put("a", new Integer(100));
        hm.put("b", new Integer(200));
        hm.put("c", new Integer(300));
        hm.put("d", new Integer(400));

        // Returns Set view
        Set< Map.Entry< String, Integer> > st = hm.entrySet();

        for (Map.Entry< String, Integer> me:st)
        {
            System.out.print(me.getKey()+":");
            System.out.println(me.getValue());
        }
    }
}
```

Output:

```
a:100
b:200
c:300
d:400
```

HashMap in java

HashMap is a part of Java's collection since Java 1.2. It provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value one must know its key. HashMap is known as HashMap because it uses a technique called Hashing. [Hashing](#) is a technique of converting a large String to small String that represents the same String. A shorter value helps in indexing and faster searches. HashSet also uses HashMap internally.

Few important features of HashMap are:

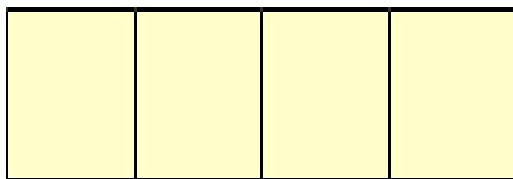
- HashMap is a part of `java.util` package.
- HashMap extends an abstract class `AbstractMap` which also provides an incomplete implementation of Map interface.
- It also implements [Cloneable](#) and [Serializable](#) interface. K and V in the above definition represent Key and Value respectively.
- HashMap doesn't allow duplicate keys but allows duplicate values. That means A single key can't contain more than 1 value but more than 1 key can contain a single value.
- HashMap allows null key also but only once and multiple null values.
- This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time. It is roughly similar to `HashTable` but is unsynchronized.
-

Internal Structure of HashMap

Internally HashMap contains an array of Node and a node is represented as a class which contains 4 fields:

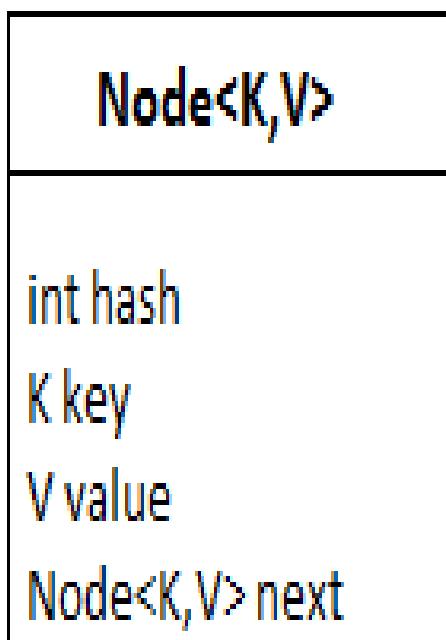
1. int hash
2. K key
3. V value
4. Node next

It can be seen that node is containing a reference of its own object. So it's a linked list.
HashMap:



Node[0]

Node:



Performance of HashMap

Performance of HashMap depends on 2 parameters:

1. Initial Capacity
2. Load Factor

Capacity is simply the number of buckets whereas the *Initial Capacity* is the capacity of HashMap instance when it is created. The *Load Factor* is a measure that when rehashing should be done. Rehashing is a process of increasing the capacity. In HashMap capacity is multiplied by 2. Load Factor is also a measure that what fraction of the HashMap is allowed to fill before rehashing. The expected number of values should be taken into account to set initial capacity. Most generally preferred load factor value is 0.75 which provides a good deal between time and space costs. Load factor's value varies between 0 and 1.

Synchronized HashMap

As it is told that HashMap is unsynchronized i.e. multiple threads can access it simultaneously. If multiple threads access this class simultaneously and at least one thread manipulates it structurally then it is necessary to make it synchronized externally. It is done by synchronizing some object which encapsulates the map. If No such object exists then it can be wrapped around Collections.synchronizedMap() to make HashMap synchronized and avoid accidental unsynchronized access. As in the following example:

```
Map m = Collections.synchronizedMap(new HashMap(...));
```

Now the Map m is synchronized.

Iterators of this class are fail-fast if any structure modification is done after the creation of iterator, in any way except through the iterator's remove method. In a failure of iterator, it will throw ConcurrentModificationException.

Constructors in HashMap

HashMap provides 4 constructors and access modifier of each is public:

1. **HashMap():** It is the default constructor which creates an instance of HashMap with initial capacity 16 and load factor 0.75.
2. **HashMap(int initial capacity):** It creates a HashMap instance with specified initial capacity and load factor 0.75.
3. **HashMap(int initial capacity, float loadFactor):** It creates a HashMap instance with specified initial capacity and specified load factor.
4. **HashMap(Map map):** It creates instance of HashMap with same mappings as specified map.

Example:

```
// Java program to illustrate  
// Java.util.HashMap  
  
import java.util.HashMap;  
import java.util.Map;  
  
public class GFG {  
    public static void main(String[] args)  
    {  
  
        HashMap<String, Integer> map
```

```
= new HashMap<>();  
  
print(map);  
map.put("vishal", 10);  
map.put("sachin", 30);  
map.put("vaibhav", 20);  
  
System.out.println("Size of map is:- "  
+ map.size());  
  
print(map);  
if (map.containsKey("vishal")) {  
    Integer a = map.get("vishal");  
    System.out.println("value for key"  
        + " \"vishal\" is:- "  
        + a);  
}  
  
map.clear();  
print(map);  
}  
  
public static void print(Map<String, Integer> map)  
{  
    if (map.isEmpty()) {  
        System.out.println("map is empty");  
    }  
  
    else {  
        System.out.println(map);  
    }  
}
```

```
    }  
}  
}
```

Output:

```
map is empty  
Size of map is:- 3  
{vaibhav=20, vishal=10, sachin=30}  
value for key "vishal" is:- 10  
map is empty
```

Time complexity of HashMap

HashMap provides constant time complexity for basic operations, get and put if the hash function is properly written and it disperses the elements properly among the buckets. Iteration over HashMap depends on the capacity of HashMap and a number of key-value pairs. Basically, it is directly proportional to the capacity + size. Capacity is the number of buckets in HashMap. So it is not a good idea to keep a high number of buckets in HashMap initially.

Methods in HashMap

1. **void clear():** Used to remove all mappings from a map.
2. **boolean containsKey(Object key):** Used to return True if for a specified key, mapping is present in the map.
3. **boolean containsValue(Object value):** Used to return true if one or more key is mapped to a specified value.
4. **Object clone():** It is used to return a shallow copy of the mentioned hash map.
5. **boolean isEmpty():** Used to check whether the map is empty or not. Returns true if the map is empty.
6. **Set entrySet():** It is used to return a set view of the hash map.
7. **Object get(Object key):** It is used to retrieve or fetch the value mapped by a particular key.
8. **Set keySet():** It is used to return a set view of the keys.
9. **int size():** It is used to return the size of a map.
10. **Object put(Object key, Object value):** It is used to insert a particular mapping of key-value pair into a map.
11. **putAll(Map M):** It is used to copy all of the elements from one map into another.
12. **Object remove(Object key):** It is used to remove the values for any particular key in the Map.
13. **Collection values():** It is used to return a Collection view of the values in the HashMap.
14. **compute(K key, BiFunction<K, V> remappingFunction):** This method Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
15. **computeIfAbsent(K key, Function<K> mappingFunction):** This method If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
16. **computeIfPresent(K key, BiFunction<K, V> remappingFunction):** This method If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.

17. **forEach(BiConsumer<K, V> action)**: This method Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
18. **getOrDefault(Object key, V defaultValue)**: This method returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
19. **merge(K key, V value, BiFunction<K, V> remappingFunction)**: This method If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
20. **putIfAbsent(K key, V value)**: This method If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
21. **replace(K key, V value)**: This method replaces the entry for the specified key only if it is currently mapped to some value.
22. **replace(K key, V oldValue, V newValue)**: This method replaces the entry for the specified key only if currently mapped to the specified value.
23. **replaceAll(BiFunction<K, V> function)**: This method replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

1. **put()**: java.util.HashMap.put() plays role in associating the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.

Syntax:

```
public V put(K key,V value)
```

Parameters:

key - key with which the specified value is to be associated

value - value to be associated with the specified key

Return: the previous value associated with

key, or null if there was no mapping for key.

2. **get()**: java.util.HashMap.get()method returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

Syntax:

```
public V get(Object key)
```

Parameters:

key - the key whose associated value is to be returned

Return: the value to which the specified

key is mapped, or null if this map contains no mapping for
the key.

3. **isEmpty():** java.util.HashMap.isEmpty() method returns true if the map contains no key-value mappings.

Syntax:

```
public boolean isEmpty()
```

Return: true if this map contains no key-value mappings

4. **size():** java.util.HashMap.size() returns the number of key-value mappings in this map.

Syntax:

```
public int size()
```

Return: the number of key-value mappings in this map.

Implementation to illustrate above methods

```
// Java program illustrating use of HashMap methods -  
  
// put(), get(), isEmpty() and size()  
  
import java.util.*;  
  
public class NewClass  
  
{  
  
    public static void main(String args[])  
  
    {  
  
        // Creation of HashMap  
        HashMap<String, String> Geeks = new HashMap<>();  
  
  
        // Adding values to HashMap as ("keys", "values")  
        Geeks.put("Language", "Java");  
        Geeks.put("Platform", "Geeks For geeks");  
        Geeks.put("Code", "HashMap");  
        Geeks.put("Learn", "More");  
  
  
        System.out.println("Testing .isEmpty() method");  
  
  
        // Checks whether the HashMap is empty or not  
        // Not empty so printing the values  
        if (!Geeks.isEmpty())
```

```

{
    System.out.println("HashMap Geeks is notempty");

    // Accessing the contents of HashMap through Keys
    System.out.println("GEEKS : " + Geeks.get("Language"));
    System.out.println("GEEKS : " + Geeks.get("Platform"));
    System.out.println("GEEKS : " + Geeks.get("Code"));
    System.out.println("GEEKS : " + Geeks.get("Learn"));

    // size() method prints the size of HashMap.
    System.out.println("Size Of HashMap : " + Geeks.size());
}

}
}

```

Output

```

Testing .isEmpty() method
HashMap Geeks is notempty
GEEKS : Java
GEEKS : Geeks For geeks
GEEKS : HashMap
GEEKS : More
Size Of HashMap : 4

```

What are the differences between HashMap and TreeMap?

1. HashMap implements Map interface while TreeMap implements SortedMap interface. A Sorted Map interface is a child of Map.
2. HashMap implements Hashing, while TreeMap implements Red-Black Tree(a Self Balancing Binary Search Tree). Therefore all [differences between Hashing and Balanced Binary Search Tree](#) apply here.
3. Both HashMap and TreeMap have their counterparts HashSet and TreeSet. HashSet and TreeSet implement [Set interface](#). In HashSet and TreeSet, we have only key, no value, these are mainly used to see presence/absence in a set. For above problem, we can't use HashSet (or TreeSet) as we can't store counts. An example problem where we would prefer HashSet (or TreeSet) over HashMap (or TreeMap) is to print all distinct elements in an array.

4. **keySet(): java.util.HashMap.keySet()** It returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.

Syntax:

```
public Set keySet()
```

Return: a set view of the keys contained in this map

5. **values(): java.util.HashMap.values()** It returns a Collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa.

6. Syntax:

```
7. public Collection values()
```

8. **Return:** a collection view of the values contained in

9. this map

10. **containsKey(): java.util.HashMap.containsKey()** It returns true if this map maps one or more keys to the specified value.

11. Syntax:

```
12. public boolean containsValue(Object value)
```

13. Parameters:

14. value - value whose presence in this map is to be tested

15. **Return:** true if this map maps one or more keys to

16. the specified value

Implementation:

```
// Java program illustrating usage of HashMap class methods  
  
// keySet(), values(), containsKey()  
  
import java.util.*;  
  
public class NewClass  
  
{  
  
    public static void main(String args[])  
  
    {  
  
        // 1 Creation of HashMap  
        HashMap<String, String> Geeks = new HashMap<>();  
  
        // 2 Adding values to HashMap as ("keys", "values")  
        Geeks.put("Language", "Java");  
        Geeks.put("Platform", "Geeks For geeks");
```

```

Geeks.put("Code", "HashMap");
Geeks.put("Learn", "More");

// 3 containsKey() method is to check the presence
// of a particluar key
// Since 'Code' key present here, the condition is true
if (Geeks.containsKey("Code"))

    System.out.println("Testing .containsKey : " +
                        Geeks.get("Code"));

// 4 keySet() method returns all the keys in HashMap
Set<String> Geekskeys = Geeks.keySet();
System.out.println("Initial keys : " + Geekskeys);

// 5 values() method return all the values in HashMap
Collection<String> Geeksvalues = Geeks.values();
System.out.println("Initial values : " + Geeksvalues);

// Adding new set of key-value
Geeks.put("Search", "JavaArticle");

// Again using .keySet() and .values() methods
System.out.println("New Keys : " + Geekskeys);
System.out.println("New Values: " + Geeksvalues);

}

}

```

Output:

```

Testing .containsKey : HashMap
Initial keys : [Language, Platform, Learn, Code]
Initial values : [Java, Geeks For geeks, More, HashMap]

```

New Keys : [Language, Platform, Search, Learn, Code]

New Values: [Java, Geeks For geeks, JavaArticle, More, HashMap]

entrySet() : `java.util.HashMap.entrySet()` method returns a complete set of keys and values present in the HashMap.

17. **Syntax:**

18. `public Set<Map.Entry> entrySet()`

19. **Return:**

20. complete set of keys and values

.getOrDefault : `java.util.HashMap.getOrDefault()` method returns a default value if there is no value find using the key we passed as an argument in HashMap. If the value for key if present already in the HashMap, it won't do anything to it.

It is very nice way to assign values to the keys that are not yet mapped, without interfering with the already present set of keys and values.

21. **Syntax:**

22. `default V getOrDefault(Object key,V defaultValue)`

23. **Parameters:**

24. key - the key whose mapped value we need to return

25. defaultValue - the default for the keys present in HashMap

26. **Return:**

27. mapping the unmapped keys with the default value.

replace() : `java.util.HashMap.replace(key, value)` or `java.util.HashMap.replace(key, oldvalue, newvalue)` method is a `java.util.HashMap` class method.

1st method accepts set of key and value which will replace the already present value of the key with the new value passed in the argument. If no such set is present replace() method will do nothing.

Meanwhile 2nd method will only replace the already present set of key-old_value if the key and old_Value are found in the HashMap.

28. **Syntax:**

29. `replace(k key, v value)`

30. or

31. `replace(k key, v oldvalue, newvalue)`

32. **Parameters:**

33. key - key in set with the old value.

34. value - new value we want to be with the specified key

35. oldvalue - old value in set with the specified key

36. newvalue - new value we want to be with the specified key

37. **Return:**

38. True - if the value is replaced

39. Null - if there is no such set present

40. **.putIfAbsent** `java.util.HashMap.putIfAbsent(key, value)` method is being used to insert a new key-value set to the HashMap if the respective set is present. Null value is returned if such key-value set is already present in the HashMap.

41. **Syntax:**

42. `public V putIfAbsent(key, value)`

43. Parameters:

44. key - key with which the specified value is associates.
45. value - value to associates with the specified key.

```
// Java Program illustrating HashMap class methods().  
  
// entrySet(), getOrDefault(), replace(), putIfAbsent  
import java.util.*;  
  
public class NewClass  
{  
  
    public static void main(String args[])  
    {  
  
        // Creation of HashMap  
        HashMap<String, String> Geeks = new HashMap<>();  
  
  
        // Adding values to HashMap as ("keys", "values")  
        Geeks.put("Language", "Java");  
        Geeks.put("Code", "HashMap");  
        Geeks.put("Learn", "More");  
  
  
        // .entrySet() returns all the keys with their values present in Hashmap  
        Set<Map.Entry<String, String>> mappingGeeks = Geeks.entrySet();  
        System.out.println("Set of Keys and Values using entrySet() : "+mappingGeeks);  
        System.out.println();  
  
  
        // Using .getOrDefault to access value  
        // Here it is Showing Default value as key - "Code" was already present  
        System.out.println("Using .getOrDefault : "  
                           + Geeks.getOrDefault("Code","javaArticle"));  
  
  
        // Here it is Showing set value as key - "Search" was not present  
        System.out.println("Using .getOrDefault : "  
                           + Geeks.getOrDefault("Search","javaArticle"));
```

```

        System.out.println();

        // .replace() method replacing value of key "Learn"
        Geeks.replace("Learn", "Methods");
        System.out.println("working of .replace() : "+mappingGeeks);
        System.out.println();

        /* .putIfAbsent() method is placing a new key-value
           as they were not present initially*/
        Geeks.putIfAbsent("cool", "HashMap methods");
        System.out.println("working of .putIfAbsent() : "+mappingGeeks);

        /* .putIfAbsent() method is not doing anything
           as key-value were already present */
        Geeks.putIfAbsent("Code", "With_JAVA");
        System.out.println("working of .putIfAbsent() : "+mappingGeeks);

    }

}


```

Output:

```
Set of Keys and Values using entrySet() : [Language=Java, Learn=More, Code=HashMap]
```

```
Using .getOrDefault : HashMap
```

```
Using .getOrDefault : javaArticle
```

```
working of .replace() : [Language=Java, Learn=Methods, Code=HashMap]
```

```
working of .putIfAbsent() : [Language=Java, cool=HashMap methods, Learn=Methods, Code=HashMap]
```

```
working of .putIfAbsent() : [Language=Java, cool=HashMap methods, Learn=Methods, Code=HashMap]
```

remove(Object key): Removes the mapping for this key from this map if present.

```
// Java Program illustrating remove() method using Iterator.

import java.util.*;
public class NewClass
{
    public static void main(String args[])
    {
        // Creation of HashMap
        HashMap<String, String> Geeks = new HashMap<>();

        // Adding values to HashMap as ("keys", "values")
        Geeks.put("Language", "Java");
        Geeks.put("Platform", "Geeks For geeks");
        Geeks.put("Code", "HashMap");

        // .entrySet() returns all the keys with their values present in Hashmap
        Set<Map.Entry<String, String>> mappingGeeks = Geeks.entrySet();
        System.out.println("Set of Keys and Values : "+mappingGeeks);
        System.out.println();

        // Creating an iterator
        System.out.println("Use of Iterator to remove the sets.");
        Iterator<Map.Entry<String, String>> geeks_iterator = Geeks.entrySet().iterator();
        while(geeks_iterator.hasNext())
        {
            Map.Entry<String, String> entry = geeks_iterator.next();
            // Removing a set one by one using iterator
            geeks_iterator.remove(); // right way to remove entries from Map,
```

```

        // avoids ConcurrentModificationException
        System.out.println("Set of Keys and Values : "+mappingGeeks);

    }

}
}

```

Output:

```
Set of Keys and Values : [Language=Java, Platform=Geeks For geeks,
Code=HashMap]
```

Use of Iterator to remove the sets.

```
Set of Keys and Values : [Platform=Geeks For geeks, Code=HashMap]
```

```
Set of Keys and Values : [Code=HashMap]
```

```
Set of Keys and Values : []
```

Advantage:

If we use for loop, it get translated to Iterator internally but without using Iterator explicitly we can't remove any entry during Iteration. On doing so, Iterator may throw ConcurrentModificationException. So, we use explicit Iterator and while loop to traverse.

LinkedHashMap in java

LinkedHashMap is just like [HashMap](#) with an additional feature of maintaining an order of elements inserted into it. HashMap provided the advantage of quick insertion, search and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order. Few important features of LinkedHashMap are as follows:

- A LinkedHashMap contains values based on the key. It implements the Map interface and extends HashMap class.
- It contains only unique elements (See [this](#) for details)..
- It may have one null key and multiple null values (See [this](#) for details).
- It is same as HashMap with additional feature that it maintains insertion order. For example, when we ran the code with HashMap, we got different order of elements (See [this](#)).

Declaration:

```
LinkedHashMap<Integer, String> lhm = new LinkedHashMap<Integer, String>();
```

Constructors in LinkedHashMap:

The LinkedHashMap accepts five types of constructors:

1. **LinkedHashMap()**: This is used to construct a default LinkedHashMap constructor.
2. **LinkedHashMap(int capacity)**: It is used to initialize a particular LinkedHashMap with a specified capacity.
3. **LinkedHashMap(Map m_a_p)**: It is used to initialize a particular LinkedHashMap with the elements of the specified map.
4. **LinkedHashMap(int capacity, float fillRatio)**: It is used to initialize both the capacity and fill ratio for a LinkedHashMap.
5. **LinkedHashMap(int capacity, float fillRatio, boolean Order)**: This constructor is also used to initialize both the capacity and fill ratio for a LinkedHashMap along with whether to follow the insertion order or not.
 - True is passed for last access order.
 - False is passed for insertion order.

Basic **Operations** of LinkedHashMap class:

```
// Java program to demonstrate working of LinkedHashMap  
import java.util.*;  
  
public class BasicLinkedHashMap  
{  
    public static void main(String a[])  
    {  
        LinkedHashMap<String, String> lhm =  
            new LinkedHashMap<String, String>();  
        lhm.put("one", "practice.geeksforgeeks.org");  
        lhm.put("two", "code.geeksforgeeks.org");  
        lhm.put("four", "quiz.geeksforgeeks.org");  
  
        // It prints the elements in same order  
        // as they were inserted  
        System.out.println(lhm);
```

```

        System.out.println("Getting value for key 'one': "
                + lhm.get("one"));

        System.out.println("Size of the map: " + lhm.size());

        System.out.println("Is map empty? " + lhm.isEmpty());

        System.out.println("Contains key 'two'? " +
                lhm.containsKey("two"));

        System.out.println("Contains value 'practice.geeks"
                +"forgeeks.org'? " + lhm.containsValue("practice" +
                ".geeksforgeeks.org"));

        System.out.println("delete element 'one': " +
                lhm.remove("one"));

        System.out.println(lhm);

    }

}

```

Output:

```

{one=practice.geeksforgeeks.org, two=code.geeksforgeeks.org,
four=quiz.geeksforgeeks.org}

Getting value for key 'one': practice.geeksforgeeks.org

Size of the map: 3

Is map empty? false

Contains key 'two'? true

Contains value 'practice.geeksforgeeks.org'? true

delete element 'one': practice.geeksforgeeks.org

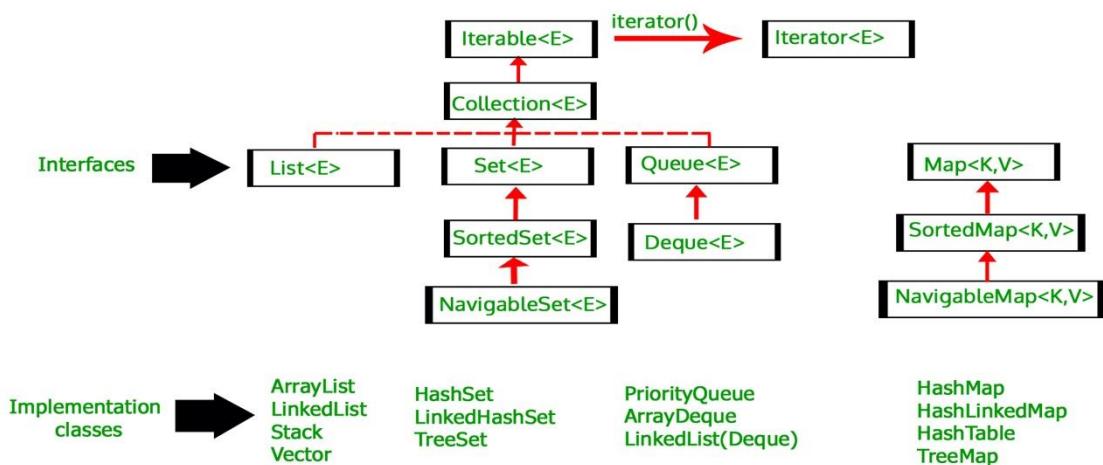
{two=code.geeksforgeeks.org, four=quiz.geeksforgeeks.org}

```

Methods in LinkedHashMap:

1. [void clear\(\)](#): This method is used to remove all the mappings from the map.
2. [boolean containsKey\(Object key\)](#): This method is used to returns true if a specified element is mapped by one or more keys.
3. [Object get\(Object key\)](#): The method is used to retrieve or fetch the value mapped by the specified key.

4. `protected boolean removeEldestEntry(Map.Entry eldest)`: The method is used to return true when the map removes its eldest entry from the map.
5. `entrySet?()`: This method returns a Set view of the mappings contained in this map.
6. `forEach?(BiConsumer<K,V> action)`: This method Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
7. `getOrDefault?(Object key, V defaultValue)`: This method returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
8. `keySet?()`: This method returns a Set view of the keys contained in this map.
9. `removeEldestEntry?(Map.Entry<K,V> eldest)`: This method returns true if this map should remove its eldest entry.
10. `replaceAll?(BiFunction<K,V> function)`: This method replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
11. `values?()`: This method returns a Collection view of the values contained in this map.



Java LinkedHashMap Example: Key-Value pair

```

1. import java.util.*;
2. class LinkedHashMap2{
3.   public static void main(String args[]){
4.     LinkedHashMap<Integer, String> map = new LinkedHashMap<Integer, String>();
5.     map.put(100,"Amit");
6.     map.put(101,"Vijay");
7.     map.put(102,"Rahul");
8.     //Fetching key
9.     System.out.println("Keys: "+map.keySet());
10.    //Fetching value
11.    System.out.println("Values: "+map.values());
12.    //Fetching key-value pair
  
```

```
13.         System.out.println("Key-Value pairs: "+map.entrySet());
14.     }
15. }
Keys: [100, 101, 102]
Values: [Amit, Vijay, Rahul]
Key-Value pairs: [100=Amit, 101=Vijay, 102=Rahul]
```

```
1. import java.util.*;
2. class Book {
3. int id;
4. String name,author,publisher;
5. int quantity;
6. public Book(int id, String name, String author, String publisher, int quantity) {
7.     this.id = id;
8.     this.name = name;
9.     this.author = author;
10.    this.publisher = publisher;
11.    this.quantity = quantity;
12. }
13. }
14. public class MapExample {
15. public static void main(String[] args) {
16.     //Creating map of Books
17.     Map<Integer,Book> map=new LinkedHashMap<Integer,Book>();
18.     //Creating Books
19.     Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20.     Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21.     Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
22.     //Adding Books to map
23.     map.put(2,b2);
24.     map.put(1,b1);
25.     map.put(3,b3);
26.
27.     //Traversing map
28.     for(Map.Entry<Integer, Book> entry:map.entrySet()){
29.         int key=entry.getKey();
30.         Book b=entry.getValue();
31.         System.out.println(key+" Details:");
32.         System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33.     }
```

```
34. }
35. }
```

Output:

```
2 Details:
102 Data Communications & Networking Forouzan Mc Graw Hill 4
1 Details:
101 Let us C Yashwant Kanetkar BPB 8
3 Details:
103 Operating System Galvin Wiley 6
```

Multidimensional Arrays in Java:

- Multidimensional Arrays can be defined in simple words as array of arrays. Data in multidimensional arrays are stored in tabular form.
- You can not specify an array dimension after an empty dimension while creating multidimensional arrays. It gives compile time error.

```
data_type[1st dimension][2nd dimension][][]..[Nth dimension] array_name = new  
data_type[size1][size2]....[sizeN];
```

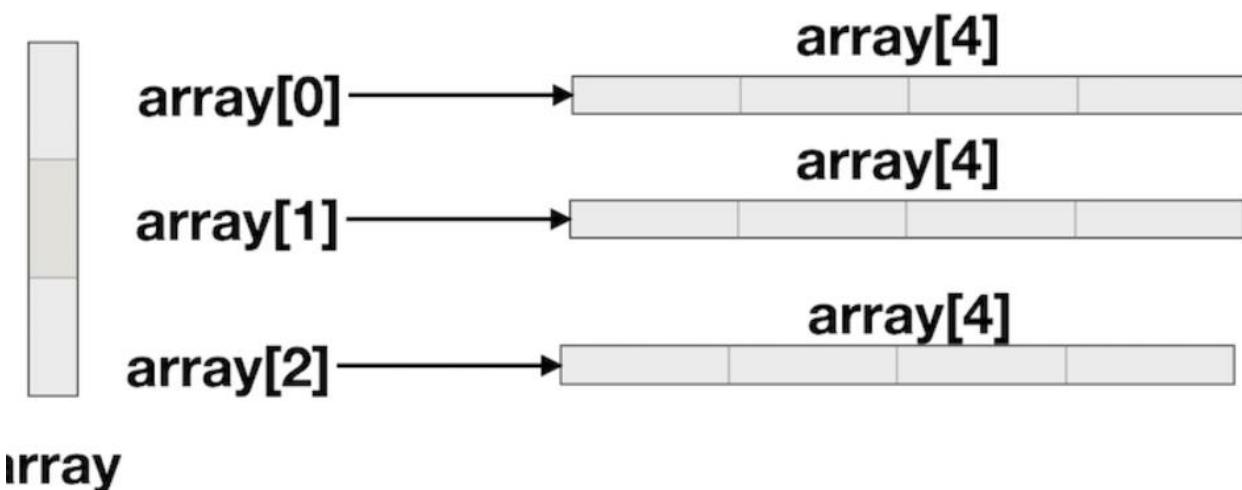
Two dimensional array:

- A two-dimensional array is an array which technically has one row of elements, however, each row has a bunch of elements defined by itself.
- It is also called as array of arrays.

```
int[][] twoD_arr = new int[3][4];
```

	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Internally declared as,



Ex:

```
public class Main {  
  
    public static void main(String[] args) {  
  
        // creating and initializing two dimensional array with shortcut syntax  
        int[][] arrInt = { { 1, 2 }, { 3, 4, 5 } };  
        for (int i = 0; i < arrInt.length; i++)  
        {  
            for (int j = 0; j < arrInt[i].length; j++)  
            {  
                System.out.print(arrInt[i][j] + " ");  
            }  
            System.out.println("");  
        }  
        System.out.println(arrInt.length);  
        System.out.println(arrInt[1].length);  
  
        String[][] arrStr = new String[3][4];  
        for (int i = 0; i < arrStr.length; i++)  
        {  
            for (int j = 0; j < arrStr[i].length; j++)  
            {  
                arrStr[i][j] = "Str" + j;  
                System.out.print(arrStr[i][j] + " ");  
            }  
            System.out.println("");  
        }  
    }  
}
```

```

}

int[][] arr = new int[2][3];

for (int k = 0; k < arr.length; k++)
{
    for (int l = 0; l < arr[k].length; l++)
    {
        arr[k][l] = l;
        System.out.print(arr[k][l] + " ");
    }
    System.out.println("");
}
}

```

O/P:

1 2

3 4 5

2

3

Str0 Str1 Str2 Str3

Str0 Str1 Str2 Str3

Str0 Str1 Str2 Str3

0 1 2

0 1 2

Three dimensional array:

- Three dimensional array is an array of arrays of arrays.
- data_type[][][] array_name = {

```

    {
        {valueA1R1C1, valueA1R1C2, ....},
    }
}
```

```

    {valueA1R2C1, valueA1R2C2, ....}
},
{
    {valueA2R1C1, valueA2R1C2, ....},
    {valueA2R2C1, valueA2R2C2, ....}
}
};

```

For example: int[][][] arr = { {{1, 2}, {3, 4}}, {{5, 6}, {7, 8}} };

Ex:

```
int[][][] threeD_arr = new int[10][20][30];
```

- The total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.
 twoD_arr can store a total of $(3 \times 4) = 12$ elements
 threeD_arr can store a total of $(10 \times 20 \times 30) = 600$ elements.

Multi dimensional array

- A multi-dimensional array is an array with more than one level or dimension, where each array length is fixed. For example, a 2D array, or two-dimensional array, is an array of arrays, meaning it is a matrix of rows and columns (think of a table). A 3D array adds another dimension, turning it into an array of arrays of arrays.

Jagged Array:

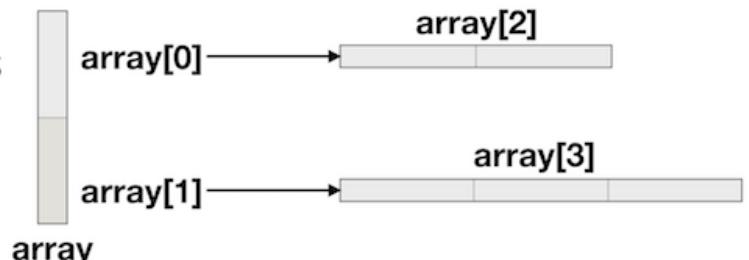
- Jagged array is array of arrays such that member arrays can be of different sizes, i.e., we can create a 2-D arrays but with variable number of columns in each row. These type of arrays are also known as Jagged arrays.

```

int[][] array = new int[2][];

array[0] = new int[2];
array[1] = new int[3];

```



Ex:

```
public class JaggedArrayExample {  
  
    public static void main(String[] args) {  
  
        // showing multidimensional arrays initializing  
        int[][] arrMulti = new int[2][]; // yes it's valid  
        int[][] arr = new int[][]{ }; //not valid;  
  
        arrMulti[0] = new int[2];  
        arrMulti[1] = new int[3];  
  
        arrMulti[0][0] = 1;  
        arrMulti[0][1] = 2;  
        arrMulti[1][0] = 3;  
        arrMulti[1][1] = 4;  
        arrMulti[1][2] = 5;  
        for (int i = 0; i < arrMulti.length; i++) {  
            for (int j = 0; j < arrMulti[i].length; j++) {  
                System.out.print(arrMulti[i][j] + " ");  
            }  
            System.out.println("");  
        }  
    }  
}
```

O/P:

```
1 2  
3 4 5
```

Array Copy in Java:

- Given an array, we need to copy its elements in a different array.
- However, from the below code new array reference will point to old array location but not copy the elements.

```
// A Java program to demonstrate that simply assigning one array  
// reference is incorrect.  
public class Test
```

```

{
    public static void main(String[] args)
    {
        int a[] = {1, 8, 3};

        // Create an array b[] of same size as a[]
        int b[] = new int[a.length];

        // Doesn't copy elements of a[] to b[], only makes
        // b refer to same location
        b = a;

        // Change to b[] will also reflect in a[] as 'a' and
        // 'b' refer to same location.
        b[0]++;

        System.out.println("Contents of a[] ");
        for (int i=0; i<a.length; i++)
            System.out.print(a[i] + " ");

        System.out.println("\n\nContents of b[] ");
        for (int i=0; i<b.length; i++)
            System.out.print(b[i] + " ");
    }
}

```

Output:

```
Contents of a[]
```

```
2 8 3
```

```
Contents of b[]
```

```
2 8 3
```

- Second way is,
- We might iterate each element of the given original array and copy one element at a time. Using this method guarantees that any modifications to b, will not alter the original array a.

Ex:

```
// A Java program to demonstrate copying by one by one
// assigning elements of a[] to b[].
public class Test
{
    public static void main(String[] args)
```

```

{
    int a[] = {1, 8, 3};

    // Create an array b[] of same size as a[]
    int b[] = new int[a.length];

    // Copy elements of a[] to b[]
    for (int i=0; i<a.length; i++)
        b[i] = a[i];

    // Change b[] to verify that b[] is different
    // from a[]
    b[0]++;
}

System.out.println("Contents of a[] ");
for (int i=0; i<a.length; i++)
    System.out.print(a[i] + " ");

System.out.println("\n\nContents of b[] ");
for (int i=0; i<b.length; i++)
    System.out.print(b[i] + " ");
}
}

```

Output:

Contents of a[]

1 8 3

Contents of b[]

2 8 3

- Third way is,
- We can use clone method in Java.
- Using this method, any modifications applied on b[] cannot alter the original array a[].

Ex:

// A Java program to demonstrate array copy using clone()

public class Test

{

 public static void main(String[] args)

{

 int a[] = {1, 8, 3};

 // Copy elements of a[] to b[]

```

int b[] = a.clone();

// Change b[] to verify that b[] is different
// from a[]
b[0]++;

System.out.println("Contents of a[] ");
for (int i=0; i<a.length; i++)
    System.out.print(a[i] + " ");

System.out.println("\n\nContents of b[] ");
for (int i=0; i<b.length; i++)
    System.out.print(b[i] + " ");
}
}

```

Output:

```
Contents of a[]
```

```
1 8 3
```

```
Contents of b[]
```

```
2 8 3
```

- And the fourth way is,
- We can also use `System.arraycopy()` Method. `System` is present in `java.lang` package.

Syntax:

```
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length).
```

Where `src` denotes the source array, `srcPos` is the index from which copying starts. Similarly, `dest` denotes the destination array, `destPos` is the index from which the copied elements are placed in the destination array. `length` is the length of subarray to be copied.

Ex:

```
// A Java program to demonstrate array copy using
// System.arraycopy()
public class Test
{
    public static void main(String[] args)
    {
        int a[] = {1, 8, 3};

        // Create an array b[] of same size as a[]
    }
}
```

```

int b[] = new int[a.length];

// Copy elements of a[] to b[]
System.arraycopy(a, 0, b, 0, 3);

// Change b[] to verify that b[] is different
// from a[]
b[0]++;

System.out.println("Contents of a[] ");
for (int i=0; i<a.length; i++)
    System.out.print(a[i] + " ");

System.out.println("\n\nContents of b[] ");
for (int i=0; i<b.length; i++)
    System.out.print(b[i] + " ");
}
}

```

Output:

```

Contents of a[]
1 8 3

Contents of b[]
2 8 3

```

Summary:

- Simply assigning reference is wrong
- Array can be copied by iterating over array, and one by one assigning elements.
- We can avoid iteration over elements using clone() or System.arraycopy()
- clone() creates a new array of same size, but System.arraycopy() can be used to copy from a source range to a destination range.
- System.arraycopy() is faster than clone() as it uses Java Native Interface

Array set() method in Java:

- The `java.lang.reflect.Array.set()` is an inbuilt method in Java and is used to set a specified value to a specified index of a given object array.

Syntax:

```
Array.set(Object []array, int index, Object value)
```

Ex:

```
import java.lang.reflect.Array;  
  
public class Main  
{  
  
    public static void main(String[] args)  
    {  
  
        // Declaring and defining an int array  
        int a[] = { 1, 2, 3, 4, 5 };  
  
        Array.set(a, 3,10);  
  
        // Printing the value  
        for(int val:a)  
  
            System.out.println(val);  
  
    }  

```

O/P:

```
1  
2  
3  
10  
5
```

Default array values in Java:

- If we don't assign values to array elements, and try to access them, compiler does not produce error as in case of simple variables. Instead it assigns values which aren't garbage.
- Below are the default assigned values.

boolean : false

```
int : 0  
double : 0.0  
String : null  
User Defined Type : null
```

NON ACCESS MODIFIERS IN JAVA

Non-access modifiers : In java, we have 7 non-access modifiers. They are used with classes, methods, variables, constructors etc to provide information about their behavior to JVM. They are

- static
- final
- abstract
- synchronized
- transient
- volatile
- native

static keyword in java

static is a non-access modifier in Java which is applicable for the following:

1. blocks
2. variables
3. methods
4. nested classes

To create a static member(block, variable, method, nested class), 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. For example, in below java program, we are accessing static method *m1()* without creating any object of *Test* class.

```
// Java program to demonstrate that a static member  
// can be accessed before instantiating a class  
class Test  
{  
    // static method  
    static void m1()  
    {  
        System.out.println("from m1");  
    }  
}
```

```
public static void main(String[] args)
{
    // calling m1 without creating
    // any object of class Test
    m1();
}
Output:
```

from m1

Static blocks

If you need to do computation in order to initialize your **static variables**, you can declare a static block that gets executed exactly once, when the class is first loaded. Consider the following java program demonstrating use of static blocks.

```
// Java program to demonstrate use of static blocks
class Test
{
    // static variable
    static int a = 10;
    static int b;

    // static block
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    public static void main(String[] args)
    {
        System.out.println("from main");
        System.out.println("Value of a : "+a);
        System.out.println("Value of b : "+b);
    }
}
Output:
```

Static block initialized.

from main

Value of a : 10

Value of b : 40

Example with Static and non static blocks

```
class Main
{
    // static variable
    static int a = 10;
    static int b;

    // static block
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    {
        System.out.println("non static block");
    }
    Main()
    {
        System.out.println("constructor");
    }
    public static void main(String[] args)
    {
        System.out.println("from main");
        Main main= new Main();
        System.out.println("Value of a : "+a);
        System.out.println("Value of b : "+b);
    }
}
```

Output:

```
Static block initialized.

from main

non static block

constructor

Value of a : 10

Value of b : 40
```

Static variables

When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

Important points for static variables :-

- We can create static variables at class-level only.
- static block and static variables are executed in order they are present in a program.

Below is the java program to demonstrate that static block and static variables are executed in order they are present in a program.

```
// java program to demonstrate execution
// of static blocks and variables
class Test
{
    // static variable
    static int a = m1();

    // static block
    static {
        System.out.println("Inside static block");
    }

    // static method
    static int m1() {
        System.out.println("from m1");
        return 20;
    }

    // static method(main !)
    public static void main(String[] args)
    {
        System.out.println("Value of a : "+a);
        System.out.println("from main");
    }
}
```

Output:

```
from m1
```

```
Inside static block
```

```
Value of a : 20
```

```
from main
```

EXAMPLE FOR STATIC LOCAL VARIABLES

```
class Main {  
    public static void main(String args[]) {  
        System.out.println(fun());  
    }  
  
    static int fun()  
    {  
        static int x= 10; //Error: Static local variables are not allowed  
        return x--;  
    }  
}
```

REFERENCE LINKS:

<https://www.geeksforgeeks.org/access-and-non-access-modifiers-in-java/>

<https://www.geeksforgeeks.org/static-keyword-java/>

NOTE:

When a method is declared with *static* keyword, it is known as static method. The most common example of a static method is *main()* method. As discussed above, Any static member can be accessed before any objects of its class are created, and without reference to any object. Methods declared as static have several restrictions:

- They can only directly call other static methods.
- They can only directly access static data.
- They cannot refer to *this* or *super* in any way.

CALLING NON STATIC METHOD IN STATIC METHOD

```
public class Main{  
    static int i=10;  
    void display()  
    {  
        System.out.println(i);  
    }  
    static void show()  
    {  
        System.out.println(i);  
        m2();  
    }  
    void m2()  
    {  
        System.out.println("from m2");  
    }  
    public static void main(String []args){  
        System.out.println("Hello World");  
        show();  
        Main hw= new Main();  
        hw.display();  
    }  
}
```

OUTPUT:

```
Main.java:18: error: non-static method m2() cannot be referenced from a static context  
t  
    m2();
```

ACCESSING NON STATIC VARIABLE IN STATIC METHOD

```
public class Main{  
    int i=10;  
    void display()  
    {  
        System.out.println(i);  
    }  
    static void show()  
    {  
        System.out.println(i);  
    }  
    public static void main(String []args){  
        System.out.println("Hello World");  
        show();  
        Main hw= new Main();  
        hw.display();  
    }  
}
```

OUTPUT:

```
Main.java:17: error: non-static variable i cannot be referenced from a static context  
    System.out.println(i);  
               ^
```

When to use static variables and methods?

Use the static variable for the property that is common to all objects. For example, in class Student, all students shares the same college name. Use static methods for changing static variables.

Consider the following java program, that illustrate the use of *static* keyword with variables and methods.

```
class Student
{
    String name;
    int rollNo;

    // static variable
    static String collName;

    // static counter to set unique roll no
    static int counter = 0;

    public Student(String name)
    {
        this.name = name;

        this.rollNo = setRollNo();
    }

    // getting unique rollNo
    // through static variable(counter)
```

```
static int setRollNo()
{
    counter++;
    return counter;
}

// static method
static void setClg(String name){
    clgName = name ;
}

// instance method
void getStudentInfo(){
    System.out.println("name : " + this.name);
    System.out.println("rollNo : " + this.rollNo);

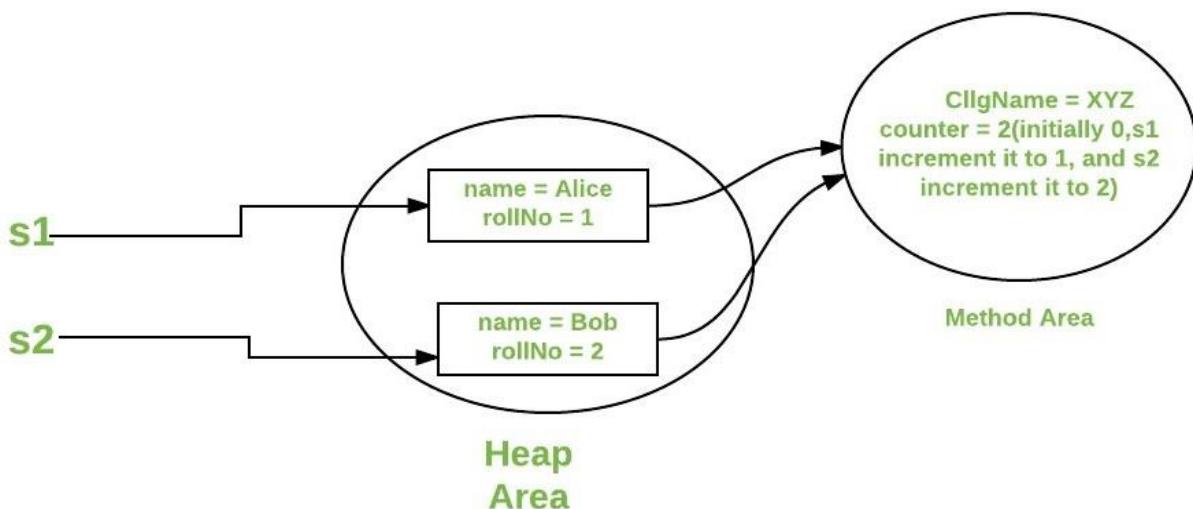
    // accessing static variable
    System.out.println("clgName : " + clgName);
}

//Driver class
public class Main
{
    public static void main(String[] args)
```

```
{  
    // calling static method  
    // without instantiating Student class  
    Student.setCllg("XYZ");  
  
    Student s1 = new Student("Alice");  
    Student s2 = new Student("Bob");  
  
    s1.getStudentInfo();  
    s2.getStudentInfo();  
  
}  
}
```

OUTPUT:

```
name : Alice  
rollNo : 1  
cllgName : XYZ  
name : Bob  
rollNo : 2  
cllgName : XYZ
```



What are the differences between static and non-static nested classes?

The following are major differences between static nested classes and inner classes.

1. A static nested class may be instantiated without instantiating its outer class.
2. Inner classes can access both static and non-static members of the outer class. A static class can access only the static members of the outer class.

```

// Java program to demonstrate how to
// implement static and non-static
// classes in a Java program.
class OuterClass {
    private static String msg = "GeeksForGeeks";

    // Static nested class
    public static class NestedStaticClass {

        // Only static members of Outer class
        // is directly accessible in nested
        // static class
        public void printMessage()
        {

            // Try making 'message' a non-static
            // variable, there will be compiler error
            System.out.println(
                "Message from nested static class: "
            )
        }
    }
}

```

```
        + msg);
    }
}

// Non-static nested class -
// also called Inner class
public class InnerClass {

    // Both static and non-static members
    // of Outer class are accessible in
    // this Inner class
    public void display()
    {
        System.out.println(
            "Message from non-static nested class: "
            + msg);
    }
}
class Main {
    // How to create instance of static
    // and non static nested class?
    public static void main(String args[])
    {
        // Create instance of nested Static class
        OuterClass.NestedStaticClass printer
            = new OuterClass.NestedStaticClass();

        // Call non static method of nested
        // static class
        printer.printMessage();

        // In order to create instance of
        // Inner class we need an Outer class
        // instance. Let us create Outer class
        // instance for creating
        // non-static nested class
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner
            = outer.new InnerClass();

        // Calling non-static method of Inner class
        inner.display();
    }
}
```

```

// We can also combine above steps in one
// step to create instance of Inner class
OuterClass.InnerClass innerObject
    = new OuterClass().new InnerClass();

// Similarly we can now call Inner class method
innerObject.display();
}
}

```

Output:

```

Message from nested static class: GeeksForGeeks
Message from non-static nested class: GeeksForGeeks
Message from non-static nested class: GeeksForGeeks

```

REFERENCE LINK:

<https://www.geeksforgeeks.org/static-class-in-java/>

final keyword in java

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

Final Variable → To create constant variables

Final Methods → Prevent Method Overriding

Final Classes → Prevent Inheritance

When a variable is declared with *final* keyword, its value can't be modified, essentially, a constant. This also means that you must initialize a final variable. If the final variable is a reference, this means that the variable cannot be re-bound

to reference another object, but internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from [final array](#) or final collection. It is good practice to represent final variables in all uppercase, using underscore to separate words.

```
// a final variable  
final int THRESHOLD = 5;  
  
// a blank final variable  
final int THRESHOLD;  
  
// a final static variable PI  
static final double PI=3.141592653589793;  
  
// a blank final static variable  
static final double PI;
```

Initializing a final variable :

We must initialize a final variable, otherwise compiler will throw compile-time error. A final variable can only be initialized once, either via an [initializer](#) or an assignment statement. There are three ways to initialize a final variable :

1. You can initialize a final variable when it is declared This approach is the most common. A final variable is called **blank final variable**, if it is **not** initialized while declaration. Below are the two ways to initialize a blank final variable.
2. A blank final variable can be initialized inside [instance-initializer block](#) or inside constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise compile time error will be thrown.
3. A blank final static variable can be initialized inside [static block](#)

```
class Gfg  
{  
    // a final variable  
    // direct initialize  
    final int THRESHOLD = 5;  
  
    // a blank final variable  
    final int CAPACITY;  
  
    // another blank final variable  
    final int MINIMUM;
```

```

// a final static variable PI
// direct initialize
static final double PI = 3.141592653589793;

// a blank final static variable
static final double EULERCONSTANT;

// instance initializer block for
// initializing CAPACITY
{
    CAPACITY = 25;
}

// static initializer block for
// initializing EULERCONSTANT
static{
    EULERCONSTANT = 2.3;
}

// constructor for initializing MINIMUM
// Note that if there are more than one
// constructor, you must initialize MINIMUM
// in them also
public GFG()
{
    MINIMUM = -1;
}
}

```

EXAMPLE FOR RE-ASSIGNING FINAL VARIABLE:

```

class Gfg
{
    static final int CAPACITY = 4;

    public static void main(String args[])
    {
        // re-assigning final variable
        // will throw compile-time error
        CAPACITY = 5;
    }
}

```

Output

Compiler Error: cannot assign a value to final variable CAPACITY

EXAMPLE FOR FINAL VARIABLE AS A REFERENCE

```
Public class Main{  
    public static void main(String args[]) {  
        final StringBuffer a=new StringBuffer("Hello");  
        System.out.println(a);  
        a.append("Welcome");  
        System.out.println(a);  
    }  
}
```

Output:

Hello

HelloWelcome

NOTE:

Once a final variable has been assigned, it always contains the same value. **If a final variable holds a reference to an object, then the state of the object may be changed by operations on the object, but the variable will always refer to the same object.**

So it is OK to manipulate state of object pointed by a
a.append("Welcome"); //is OK

but just can't reassign a with another object
final StringBuffer a = new StringBuffer("Hello");
a = new StringBuffer("World"); //this wont compile

When a final variable is created inside a method/constructor/block, it is called local final variable, and it must initialize once where it is created. See below program for local final variable

```
// Java program to demonstrate  
// local final variable  
  
// The following program compiles and runs fine  
  
class Gfg  
{
```

```
public static void main(String args[])
{
    // local final variable
    final int i;
    i = 20;
    System.out.println(i);
}
}
```

Output:

```
20
```

Final with foreach loop : final with for-each statement is a legal statement.

```
public class EnhancedForTest {
    public static void main(String... args) {
        String[] strArr = {"A", "B", "C", "D"};

        for (final String s : strArr) {
            System.out.println(s);
        }
    }
}
```

Since the String s is declared as final, this code should not compile. However, this is working fine. Why?

The enhanced for-loop as shown above is actually just as the following code:

```
public class EnhancedForTest {
    public static void main(String... args) {
        String[] strArr = {"A", "B", "C", "D"};

        for (int index = 0; index < strArr.length; index++) {
            final String s = strArr[index];
            System.out.println(s);
        }
    }
}
```

```
for(final int i=0;i<arr.length; i++) {
```

```
        System.out.println(arr[i]);
    }
```

REFERENCE LINK:

<https://www.geeksforgeeks.org/final-keyword-java/>

Final classes

When a class is declared with *final* keyword, it is called a final class. A final class cannot be extended(inherited). There are two uses of a final class :

1. One is definitely to prevent inheritance, as final classes cannot be extended.
2. The other use of final with classes is to create an immutable class like the predefined String class. You can not make a class immutable without making it final.

```
final class A
{
    // methods and fields
}

// The following class is illegal.

class B extends A
{
    // COMPILE-ERROR! Can't subclass A
}
```

Final methods

When a method is declared with *final* keyword, it is called a final method. A final method cannot be overridden. We must declare methods with final keyword for which we required to follow the same implementation throughout all the derived classes

```
class A
{
    final void m1()
```

```

{
    System.out.println("This is a final method.");
}
}

class B extends A
{
    void m1()
    {
        // COMPILE-ERROR! Can't override.
        System.out.println("Illegal!");
    }
}

```

The abstract Modifier

Abstract Class

An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.

A class cannot be both abstract and final (since a final class cannot be extended). If a class contains abstract methods then the class should be declared abstract. Otherwise, a compile error will be thrown.

An abstract class may contain both abstract methods as well normal methods.

Example

```

abstract class Caravan {
    private double price;
    private String model;
    private String year;
    public abstract void goFast(); // an abstract method
    public abstract void changeColor();
}

```

Abstract Methods

An abstract method is a method declared without any implementation. The methods body (implementation) is provided by the subclass. Abstract methods can never be final or strict.

Any class that extends an abstract class must implement all the abstract methods of the super class, unless the subclass is also an abstract class.

If a class contains one or more abstract methods, then the class must be declared abstract. An abstract class does not need to contain abstract methods.

The abstract method ends with a semicolon. Example: public abstract sample();

Example

```
public abstract class SuperClass {  
    abstract void m(); // abstract method  
}  
  
class SubClass extends SuperClass {  
    // implements the abstract method  
    void m() {  
        .....  
    }  
}
```

Important rules for abstract methods:

- Any class that contains zero or more abstract methods must also be declared abstract
- An abstract keyword cannot be used with variables and constructors, abstract keyword applicable only for class and methods.
- An abstract class can contain the main method and the final method.
- We can declare the local inner class as abstract.
- We can declare the abstract method with a throw clause.
- Abstract class can contain constructors
- The following are various **illegal combinations** of other modifiers for methods with respect to *abstract* modifier :
 1. final
 2. abstract native
 3. abstract synchronized
 4. abstract static
 5. abstract private
 6. abstract strictfp

```
abstract class A  
{  
    // abstract with method  
    // it has no body  
    abstract void m1();  
  
    // concrete methods are still allowed in abstract classes  
    void m2()  
    {
```

```

        System.out.println("This is a concrete method.");
    }
}

// concrete class B
class B extends A
{
    // class B must override m1() method
    // otherwise, compile-time exception will be thrown
    void m1() {
        System.out.println("B's implementation of m2.");
    }

}

// Driver class
public class AbstractDemo
{
    public static void main(String args[])
    {
        B b = new B();
        b.m1();
        b.m2();
    }
}

```

Output:

B's implementation of m2.

This is a concrete method.

Note : Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to [run-time polymorphism](#) is implemented through the use of super-class references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

synchronized

When a method is synchronized it can be accessed by only one thread at a time.
This can only be used to methods

Synchronization in java is implemented using **synchronized** keyword. synchronized keyword can be used with methods or blocks but not with the variables.

When a method or block is declared as synchronized, only one thread can enter into that method or block. When one thread is executing synchronized method or block, the other threads which wants to execute that method or block wait or suspend their execution until first thread is done with that method or block. Thus avoiding the thread interference and achieving thread safeness. This can be explained well with the help of an example.

Consider this example,

```
class Shared
{
    int i;

    synchronized void SharedMethod()
    {
        Thread t = Thread.currentThread();

        for(i = 0; i <= 1000; i++)
        {
            System.out.println(t.getName() + " : " + i);
        }
    }
}

public class ThreadsInJava
{
    public static void main(String[] args)
    {
        final Shared s1 = new Shared();

        Thread t1 = new Thread("Thread - 1")
        {
            @Override
            public void run()
```

```

        {
            s1.SharedMethod();
        }
    };

    Thread t2 = new Thread("Thread - 2")
    {
        @Override
        public void run()
        {
            s1.SharedMethod();
        }
    };
}

t1.start();

t2.start();
}
}

```

In the above example, both threads t1 and t2 wants to execute sharedMethod() of s1 object. But, sharedMethod() is declared as synchronized. So, whichever thread enters first into sharedMethod(), it continues to execute that method. The other thread waits for first thread to finish it's execution of sharedMethod(). It never enters into sharedMethod() until first thread is done with that method. That means, both threads are executing sharedMethod() one by one not simultaneously. This protects the value of "i" in the memory for a particular thread.

In synchronization, there are two types of locks on threads:

- Whenever an object is created to any class, an object lock is created and is stored inside the object.
1. **Object level lock :** Every object in java has a unique lock. Whenever we are using synchronized keyword, then only lock concept will come in the picture. If a thread wants to execute synchronized method on the given object. First, it has to get lock of that object. Once thread got the lock then it is allowed to execute any synchronized method on that object. Once method execution

completes automatically thread releases the lock. Acquiring and release lock internally is taken care by JVM and programmer is not responsible for these activities. Lets have a look on the below program to understand the object level lock:

2 .Class level lock : Every class in java has a unique lock which is nothing but class level lock. If a thread wants to execute a static synchronized method, then thread requires class level lock. Once a thread got the class level lock, then it is allowed to execute any static synchronized method of that class. Once method execution completes automatically thread releases the lock.

REFERENCE LINK:

<https://www.geeksforgeeks.org/object-level-class-level-lock-java/>

Synchronized Blocks :

Some times, you need only some part of the method to be synchronized not the whole method. This can be achieved with synchronized blocks. Synchronized blocks must be defined inside a definition blocks like methods, constructors, static initializer or instance initializer.

synchronized block takes one argument and it is called **mutex**. if synchronized block is defined inside non-static definition blocks like non-static methods, instance initializer or constructors, then this mutex must be an instance of that class. If synchronized block is defined inside static definition blocks like static methods or static initializer, then this mutex must be like `ClassName.class`.

Here is an example of static and non-static synchronized blocks.

```
1  class Shared
2  {
3      static void staticMethod()
4      {
5          synchronized (Shared.class)
6          {
7              //static synchronized block
8          }
9      }
10
11     void NonStaticMethod()
12     {
13         synchronized (this)
```

```
14      {
15          //Non-static synchronized block
16      }
17  }
18
19 void anotherNonStaticMethod()
20 {
21     synchronized (new Shared())
22     {
23         //Non-static synchronized block
24     }
25 }
26 }
```

Points-To-Remember About Synchronization In Java :

- 1) You can use **synchronized** keyword only with methods but not with variables, constructors, static initializer and instance initializers.

```
1 class Shared
2 {
3     synchronized int i; //compile time error, can't use synchronized keyword with variable
4
5     synchronized public Shared()
6     {
7         //compile time error, constructors can not be synchronized
8     }
9
10    synchronized static
11    {
12        //Compile time error, Static initializer can not be synchronized
```

```
13    }
14
15    synchronized
16    {
17        //Compile time error, Instance initializer can not be synchronized
18    }
19 }
```

2) Constructors, Static initializer and instance initializer can't be declared with synchronized keyword, but they can contain synchronized blocks.

```
1   class Shared
2   {
3       public Shared()
4       {
5           synchronized (this)
6           {
7               //synchronized block inside a constructor
8           }
9       }
10
11      static
12      {
13          synchronized (Shared.class)
14          {
15              //synchronized block inside a static initializer
16          }
17      }
```

```
18
19  {
20      synchronized (this)
21  {
22      //synchronized block inside a instance initializer
23  }
24 }
25 }
```

3) Both static and non-static methods can use synchronized keyword. For static methods, thread need class level lock and for non-static methods, thread need object level lock.

```
1   class Shared
2   {
3       synchronized static void staticMethod()
4   {
5       //static synchronized method
6   }
7
8   synchronized void NonStaticMethod()
9   {
10      //Non-static Synchronized method
11   }
12 }
```

4) It is possible that both static synchronized and non-static synchronized methods can run simultaneously. Because, static methods need class level lock and non-static methods need object level lock.

5) A method can contain any number of synchronized blocks. This is like synchronizing multiple parts of a method.

```
1  class Shared
2  {
3      static void staticMethod()
4      {
5          synchronized (Shared.class)
6          {
7              //static synchronized block - 1
8          }
9
10         synchronized (Shared.class)
11         {
12             //static synchronized block - 2
13         }
14     }
15
16     void NonStaticMethod()
17     {
18         synchronized (this)
19         {
20             //Non-static Synchronized block - 1
21         }
22
23         synchronized (this)
24         {
25             //Non-static Synchronized block - 2
26         }
27     }
28 }
```

```
26      }
27  }
28 }
```

6) Synchronization blocks can be nested.

```
1 synchronized (this)
2 {
3     synchronized (this)
4     {
5         //Nested synchronized blocks
6     }
7 }
```

7) Lock acquired by the thread before executing a synchronized method or block must be released after the completion of execution, no matter whether execution is completed normally or abnormally (due to exceptions).

8) synchronized method or block is very slow. They decrease the performance of an application. So, special care need to be taken while using synchronization. Use synchronization only when you needed it the most.

9) Use synchronized blocks instead of synchronized methods. Because, synchronizing some part of a method improves the performance than synchronizing the whole method.

REFERENCE LINK:

<https://javaconceptoftheday.com/synchronization-in-java/>

native

- "native" is a keyword which is introduced in java.

- "native" is the modifier applicable for methods only but it is not applicable for variable and classes.
- The native methods are implemented in some other language like C, C++, etc.
- The purpose of the native method is to improve the performance of the system.
- We know that the implementation of native methods is available in other languages so we don't need to care about the implementation.

The native keyword is applied to a method to indicate that the method is implemented in native code using JNI(Java Native Interface). It marks a method, that it will be implemented in other languages, not in Java.

Native methods are currently needed when

- You need to call a library from Java that is written in other language.
- You need to access system or hardware resources that are only reachable from the other language

Example: We will see the way of writing native methods

```
class Native {
    static {
        // Load Native Library
        System.loadLibrary("native library");
    }
    // Native Method Declaration
    public native void display();
}

class Main {
    public static void main(String[] args) {
        Native native = new Native();
        native.display();
    }
}
```

REFERENCE LINK:

<https://www.includehelp.com/java/what-are-the-non-access-modifiers-in-java.aspx>

transient keyword in Java

transient is a variables modifier used in **serialization**. At the time of serialization, if we don't want to save value of a particular variable in a file, then we use **transient** keyword. When JVM comes across **transient** keyword, it ignores original value of the variable and save default value of that variable data type. **transient** keyword plays an important role to meet security constraints. There are various real-life examples where we don't want to save private data in file. Another use of **transient** keyword is not to serialize the variable whose value can be calculated/derived using other serialized objects or system such as age of a person, current date, etc. Practically we serialized only those fields which represent a state of instance, after all serialization is all about to save state of an object to a file. It is good habit to use **transient** keyword with private confidential fields of a class during serialization.

```
// A sample class that uses transient keyword to
// skip their serialization.
class Test implements Serializable
{
    // Making password transient for security
    private transient String password;

    // Making age transient as age is auto-
    // computable from DOB and current date.
    transient int age;

    // serialize other fields
    private String username, email;
    Date dob;

    // other code
}
```

transient and static : Since **static** fields are not part of state of the object, there is no use/impact of using **transient** keyword with static variables. However there is no compilation error.

transient and final : final variables are directly serialized by their values, so there is no use/impact of declaring final variable as **transient**. There is no compile-time error though.

```
import java.io.*;
class Test implements Serializable
{
    // Normal variables
    int i = 10, j = 20;

    // Transient variables
    transient int k = 30;

    // Use of transient has no impact here
    transient static int l = 40;
    transient final int m = 50;

    public static void main(String[] args) throws Exception
    {
        Test input = new Test();

        // serialization
        FileOutputStream fos = new FileOutputStream("abc.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(input);

        // de-serialization
        FileInputStream fis = new FileInputStream("abc.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Test output = (Test)ois.readObject();
        System.out.println("i = " + output.i);
        System.out.println("j = " + output.j);
        System.out.println("k = " + output.k);
        System.out.println("l = " + output.l);
        System.out.println("m = " + output.m);
    }
}
```

OUTPUT:

i = 10

j = 20

k = 0

l = 40

m = 50

REFERENCE LINK:

<https://www.geeksforgeeks.org/transient-keyword-java/>

ATOMIC VARIABLES REFERENCE LINK:

<https://dzone.com/articles/java-concurrency-atomic-variables>

The Volatile Modifier

The volatile modifier is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.

Accessing a volatile variable synchronizes all the cached copies of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

Example

```
public class MyRunnable implements Runnable {  
    private volatile boolean active;  
  
    public void run() {  
        active = true;  
        while (active) { // line 1  
            // some code here  
        }  
    }  
  
    public void stop() {  
        active = false; // line 2  
    }  
}
```

Usually, run() is called in one thread (the one you start using the Runnable), and stop() is called from another thread. If in line 1, the cached value of active is used, the loop may not stop when you set active to false in line 2. That's when you want to use *volatile*.

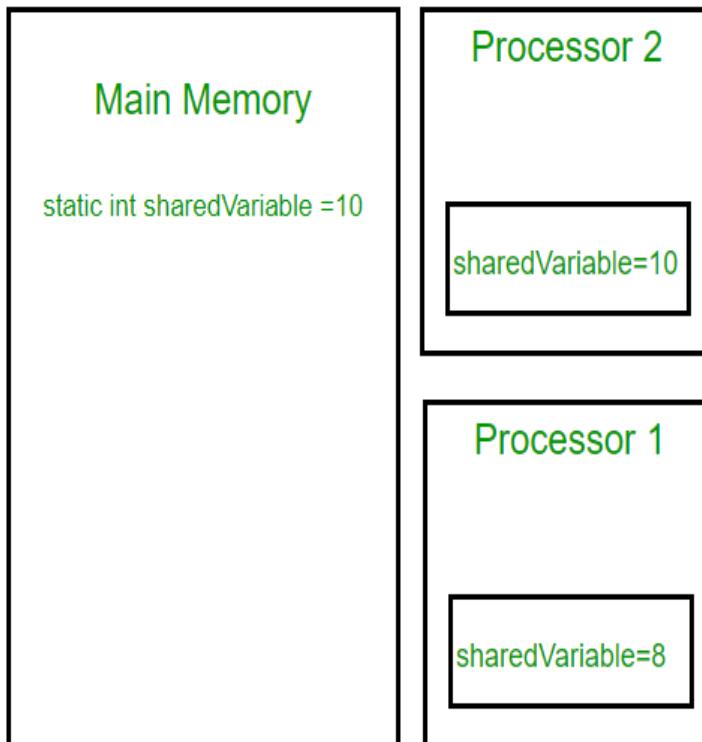
Using volatile is yet another way (like synchronized, atomic wrapper) of making class thread safe. Thread safe means that a method or class instance can be used by multiple threads at the same time without any problem.

Consider below simple example.

```
class SharedObj
{
    // Changes made to sharedVar in one thread
    // may not immediately reflect in other thread
    static int sharedVar = 6;
}
```

Suppose that two threads are working on **SharedObj**. If two threads run on different processors each thread may have its own local copy of **sharedVariable**. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. This depends on the [write policy](#) of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.

Below diagram shows that if two threads are run on different processors, then value of **sharedVariable** may be different in different threads.



Note that write of normal variables without any synchronization actions, might not be visible to any reading thread (this behavior is called [sequential consistency](#)).

Although most modern hardware provide good cache coherence therefore most probably the changes in one cache are reflected in other but it's not a good practice to rely on hardware for to 'fix' a faulty application.

```
class SharedObj
{
    // volatile keyword here makes sure that
    // the changes made in one thread are
    // immediately reflect in other thread
    static volatile int sharedVar = 6;
}
```

Note that volatile should not be confused with static modifier. static variables are class members that are shared among all objects. There is only one copy of them in main memory.

volatile vs synchronized:

Before we move on let's take a look at two important features of locks and synchronization.

1. **Mutual Exclusion:** It means that only one thread or process can execute a block of code (critical section) at a time.
2. **Visibility:** It means that changes made by one thread to shared data are visible to other threads.

Java's synchronized keyword guarantees both mutual exclusion and visibility. If we make the blocks of threads that modifies the value of shared variable synchronized only one thread can enter the block and changes made by it will be reflected in the main memory. All other thread trying to enter the block at the same time will be blocked and put to sleep.

In some cases we may only desire the visibility and not atomicity. Use of synchronized in such situation is an overkill and may cause scalability problems. Here volatile comes to the rescue. Volatile variables have the visibility features of synchronized but not the atomicity features. The values of volatile variable will never be cached and all writes and reads will be done to and from the main memory. However, use of volatile is limited to very restricted set of cases as most of the times atomicity is desired. For example a simple increment statement such as $x = x + 1;$ or $x++$ seems to be a single operation but is s really a compound read-modify-write sequence of operations that must execute atomically.

strictfp

strictfp is a keyword in the Java programming language that restricts floating-point calculations to ensure portability. The strictfp command was introduced into Java with the Java virtual machine (JVM) version 1.2 and is available for use on all currently updated Java VMs.

Strictfp ensures that you get exactly the same results from your floating point calculations on every platform. If you don't use strictfp, the JVM implementation is free to use extra precision where available.

strictfp can be used on classes, interfaces and non-abstract methods. When applied to a method, it causes all calculations inside the method to use strict floating-point math. When applied to a class, all calculations inside the class use strict floating-point math.

```
public strictfp class MyFPclass {  
    // ... contents of class here ...  
}
```

OBJECT CLASS AND ITS METHODS

Object class is in the default package i.e java.lang package .

The Object class defines the basic state and behavior that all objects must have, such as the ability to compare oneself to another object, to convert to a string, to wait on a condition variable, to notify other objects that a condition variable has changed, and to return the object's class.

Every class in Java is directly or indirectly derived from the **Object** class. If a Class does not extend any other class then it is direct child class of **Object** and if extends other class then it is an indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

METHODS OF OBJECT CLASS:

toString() : `toString()` provides String representation of an Object and used to convert an object to String. The default `toString()` method for class Object returns a string consisting of the name of the class of which the object is an instance, the

at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object.

It is always recommended to override **toString()** method to get our own String representation of Object.

equals(Object obj) : Compares the given object to "this" object (the object on which the method is called). It gives a generic way to compare objects for equality. It is recommended to override **equals(Object obj)** method to get our own equality condition on Object.

Note : It is generally necessary to override the **hashCode()** method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

hashCode() : For every object, JVM generates a unique number which is hashCode. It returns distinct integers for distinct objects. A common misconception about this method is that hashCode() method returns the address of object, which is not correct. It converts the internal address of object to an integer by using an algorithm. The hashCode() method is **native** because in Java it is impossible to find address of an object, so it uses native languages like C/C++ to find address of the object.

- Multiple invocations of hashCode() should return the same integer value, unless the object property is modified that is being used in the equals() method.
- An object hash code value can change in multiple executions of the same application.
- If two objects are equal according to equals() method, then their hash code must be same.
- If two objects are unequal according to equals() method, their hash code are not required to be different. Their hash code value may or may-not be equal.

If o1.equals(o2), then o1.hashCode() == o2.hashCode() should always be true.

If o1.hashCode() == o2.hashCode is true, it doesn't mean that o1.equals(o2) will be true.

When to override equals() and hashCode() methods?

When we override equals() method, it's almost necessary to override the hashCode() method too so that their contract is not violated by our implementation.

Note that your program will not throw any exceptions if the equals() and hashCode() contract is violated, if you are not planning to use the class as Hash table key, then it will not create any problem.

If you are planning to use a class as Hash table key, then it's must to override both equals() and hashCode() methods.

REFERENCE LINK:

<https://www.journaldev.com/21095/java-equals-hashcode>

getClass()

getClass() is the method of Object class. This method returns the runtime class of this object. The class object which is returned is the object that is locked by static synchronized method of the represented class.

Syntax

1. **public final Class<?> getClass()**

Returns

It returns the Class objects that represent the runtime class of this object.

finalize() method : This method is called just before an object is garbage collected. It is called by the [Garbage Collector](#) on an object when garbage collector determines that there are no more references to the object. We should override finalize() method to dispose system resources, perform clean-up activities and minimize memory leaks. For example before destroying Servlet objects web container, always called finalize method to perform clean-up activities of the session.

Note :finalize method is called just **once** on an object even though that object is eligible for garbage collection multiple times.

```
// Java program to demonstrate working of finalize()
public class Test
{
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println(t.hashCode());

        t = null;

        // calling garbage collector
        System.gc();
```

```
        System.out.println("end");
    }

    @Override
    protected void finalize()
    {
        System.out.println("finalize method called");
    }
}
```

Output:

```
366712642
finalize method called
end
```

1. wait()

It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls `notify()`. The `wait()` method releases the lock prior to waiting and reacquires the lock prior to returning from the `wait()` method. The `wait()` method is actually tightly integrated with the synchronization lock, using a feature not available directly from the synchronization mechanism.

In other words, it is not possible for us to implement the `wait()` method purely in Java. It is a **native method**.

General syntax for calling `wait()` method is like this:

`wait()` method syntax

```
synchronized( lockObject )
{
    while( ! condition )
    {
        lockObject.wait();
    }
}
```

```
//take the action here;
```

```
}
```

2. notify()

It wakes up one single thread that called `wait()` on the same object. It should be noted that calling `notify()` does not actually give up a lock on a resource. It tells a waiting thread that that thread can wake up. However, the lock is not actually given up until the notifier's synchronized block has completed.

So, if a notifier calls `notify()` on a resource but the notifier still needs to perform 10 seconds of actions on the resource within its synchronized block, the thread that had been waiting will need to wait at least another additional 10 seconds for the notifier to release the lock on the object, even though `notify()` had been called.

General syntax for calling `notify()` method is like this:

`notify()` method syntax

```
synchronized(lockObject)
```

```
{
```

```
    //establish_the_condition;
```

```
    lockObject.notify();
```

```
    //any additional code if needed
```

```
}
```

3. notifyAll()

It wakes up all the threads that called `wait()` on the same object. The highest priority thread will run first in most of the situation, though not guaranteed. Other things are same as `notify()` method above.

General syntax for calling `notify()` method is like this:

`notifyAll()` method syntax

```
synchronized(lockObject)
```

```
{  
    establish_the_condition;  
  
    lockObject.notifyAll();  
}
```

OPERATORS

UNARY OPERATORS:

INCREMENT AND DECREMENT:

```
class OperatorExample{  
    public static void main(String args[]){  
        int a=10;  
  
        System.out.println(-a);//-10  
        System.out.println(+a);//10  
        System.out.println(++a);//11  
        System.out.println(a);//10  
        System.out.println(--a)//9  
    }  
}  
  
class OperatorExample{  
    public static void main(String args[]){  
        int a=10;  
  
        int b=10;  
  
        System.out.println(a++ + ++a);//10+12=22  
        System.out.println(b++ + b++);//10+11=21  
    }  
}
```

NOT OPERATION :

only for booleans

```
class OperatorExample{  
    public static void main(String args[]){  
        boolean c=true;  
        boolean d=false;  
        System.out.println(!c);//false (opposite of boolean value)  
        System.out.println(!d);//true  
    }  
}
```

BITWISE OPERATORS (&, |, ^, ~, <<, >>, >>>)

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows –

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

```
a^b = 0011 0001
```

```
~a = 1100 0011
```

```
System.out.println(4&5)//4 4-->100 and 5-->101
```

```
100
```

```
101
```

```
---
```

```
100==>4
```

```
System.out.println(4|5);//5
```

```
System.out.println(4^5);//1
```

SHIFT OPERATORS:

LEFT SHIFT:

```
class OperatorExample{  
    public static void main(String args[]){  
        System.out.println(10<<2);//10*2^2=10*4=40 10-->00001010 --> 00101000=40  
        System.out.println(10<<3);//10*2^3=10*8=80 10-->00001010-->01010000=80  
        System.out.println(20<<2);//20*2^2=20*4=80  
        System.out.println(15<<4);//15*2^4=15*16=240  
        System.out.println(-5<<2);//-5*2^2=-20  
    }
```

Explanation:

-5 is negative value so first take binary value for 5-->00000101

take 2's complement for 5-->11111011

shift left by 2 digits-->11101100(msb vdigit indicate sign)

since it is negative value take 2's complement for that binary value -->0010100==>-20

RIGHT SHIFT:

in right shift we will add zeros or ones at msb position based on the digit in msb, but when we are doing rightshift filled with zero(>>>) we will fill msb with zeroes without verifying msb digit value)

```
class OperatorExample{  
    public static void main(String args[]){  
        System.out.println(10>>2);//10/2^2=10/4=2  
        System.out.println(20>>2);//20/2^2=20/4=5  
        System.out.println(20>>3);//20/2^3=20/8=2  
        System.out.println(-4>>2);//-4/2^2=-1  
    }  
}
```

Explanation:

binary value for 10-->00001010-->shift 2 digits right will be==>00000010==>2

for negative values:

take binary value for 4-->00000100

take 2's complement ==>11111100

shift 2 digits towards right 111111 since msb is 1 we will add one on msb side

since it indicates negative value take 2's complement for that binary value ==>10000001==>-1

take 2's complement for 4 will be binary value for -4 shift

RIGHT SHIFT FILLED WITH ZEROS:

```
class OperatorExample{  
    public static void main(String args[]){  
        //For positive number, >> and >>> works same  
        System.out.println(20>>2);  
        System.out.println(20>>>2);  
        //For negative number, >>> changes parity bit (MSB) to 0  
        System.out.println(-20>>2);  
        System.out.println(-20>>>2); (while doing right shift we will fill msb position with zeros)  
    }  
}
```

```
}
```

Output:

```
5  
5  
-5  
1073741819
```

BITWISE COMPLEMENT(INVERSION OPERATION):(~):

only for integral values

```
class OperatorExample{  
    public static void main(String args[]){  
        int a=10;  
        int b=-10;  
        System.out.println(~a);//-11  
        System.out.println(~b)//9  
    }  
}
```

Explanation:

10-->00001010==>1's complement==>11110101(MSB indicating negative value so for negative value we need to take 2's complement)

so 2's complement for 11110101 is==>00001010+1==>00001011==>-11

shortcut:

for positive values the result will be reverse of + and add one from the given value

for negative values the result will be reverse of - and reduce one for given value

LOGICAL OPERATORS:

logical operators are used only for boolean values

```

public class Test {

    public static void main(String args[]) {
        boolean a = true;
        boolean b = false;

        System.out.println("a && b = " + (a&&b));
        System.out.println("a || b = " + (a||b) );
        System.out.println("!(a && b) = " + !(a && b));

    }
}

```

LOGICAL AND(&&) AND BITWISE AND(&):

```

class OperatorExample{
    public static void main(String args[]){
        int a=10;
        int b=5;
        int c=20;

        System.out.println(a<b&&a++<c);//false && true = false
        System.out.println(a);//10 because second condition is not checked
        System.out.println(a<b&a++<c);//false && true = false
        System.out.println(a);//11 because second condition is checked
    }
}

```

INSTANCEOF OPERATOR:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as –

(Object reference variable) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example –

Example

Live Demo

```
public class Test {  
  
    public static void main(String args[]) {  
  
        String name = "James";  
  
        // following will return true since name is type of String  
        boolean result = name instanceof String;  
        System.out.println( result );  
    }  
}
```

Output

true

```
class Vehicle {}
```

```
public class Car extends Vehicle {
```

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

```
Vehicle a = new Car();  
boolean result = a instanceof Car;  
System.out.println( result );  
}  
}
```

Output

true

TERNARY OPERATOR:

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as –

variable x = (expression) ? value if true : value if false

Following is an example –

Example

Live Demo

```
public class Test {  
  
    public static void main(String args[]) {  
        int a, b;  
        a = 10;  
        b = (a == 1) ? 20: 30;  
        System.out.println( "Value of b is : " + b );  
  
        b = (a == 10) ? 20: 30;
```

```
        System.out.println( "Value of b is : " + b );  
    }  
}
```

Output

Value of b is : 30

Value of b is : 20

REFERENCE:

<https://www.javatpoint.com/operator-shifting>

OverLoading:

Overloading occurs when two or more methods in one class have the same method name but different parameters.

OverRiding:

Overriding means having two methods with the same method name and parameters (i.e., *method signature*). One of the methods is in the parent class and the other is in the child class. Overriding allows a child class to provide a specific implementation of a method that is already provided by its parent class.

Overriding

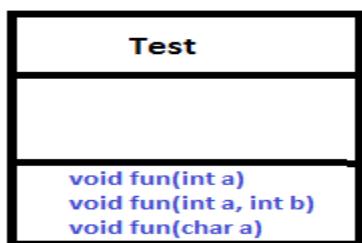
```
class Dog{  
    public void bark(){  
        System.out.println("woof ");  
    }  
}  
class Hound extends Dog{  
    public void sniff(){  
        System.out.println("sniff ");  
    }  
  
    public void bark(){  
        System.out.println("bowl");  
    }  
}
```

Same Method Name,
Same parameter

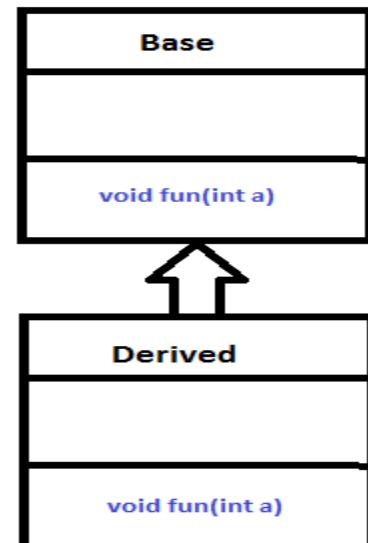
Overloading

```
class Dog{  
    public void bark(){  
        System.out.println("woof ");  
    }  
}  
//overloading method  
public void bark(int num){  
    for(int i=0; i<num; i++)  
        System.out.println("woof ");  
}
```

Same Method Name,
Different Parameter



Overloading



Overriding

EXAMPLE OF COMPILE TIME POLYMORPHISM

```
class Demo {  
    public void show(int x)  
    {  
        System.out.println("In int " + x);  
    }  
    public void show(String s)
```

```

{
    System.out.println("In String" + s);
}
public void show(byte b)
{
    System.out.println("In byte" + b);
}
}
class UseDemo {
    public static void main(String[] args)
    {
        byte a = 25;
        Demo obj = new Demo();
        obj.show(a); // it will go to
        // byte argument
        obj.show("hello"); // String
        obj.show(250); // Int
        obj.show('A'); // Since char is
        // not available, so the datatype
        // higher than char in terms of
        // range is int.
        obj.show("A"); // String
        obj.show(7.5);
    }
}

```

Can we overload main() in Java?

Like other static methods, we **can** overload main() in Java. Refer overloading main() in Java for more details.

```

// A Java program with overloaded main()
import java.io.*;

public class Test {

    // Normal main()
    public static void main(String[] args)
    {
        System.out.println("Hi Geek (from main)");
        Test.main("Geek");
    }
}
```

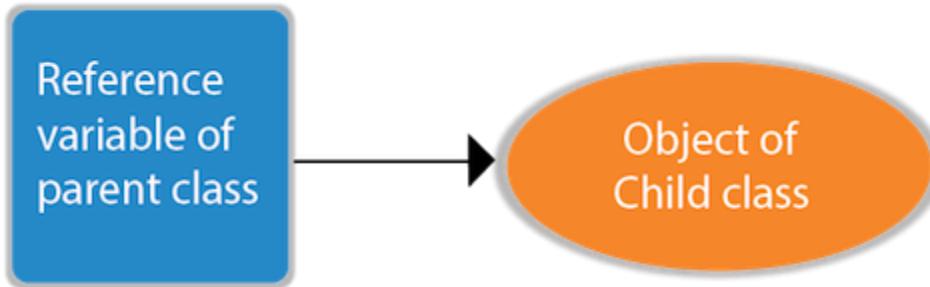
```
// Overloaded main methods
public static void main(String arg1)
{
    System.out.println("Hi, " + arg1);
    Test.main("Dear Geek", "My Geek");
}
public static void main(String arg1, String arg2)
{
    System.out.println("Hi, " + arg1 + ", " + arg2);
}
```

Output :

```
Hi Geek (from main)
Hi, Geek
Hi, Dear Geek, My Geek
```

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



1. **class** A{}
2. **class** B **extends** A{}
1. A a=**new** B(); //upcasting

For upcasting, we can use the reference variable of class type or an interface type. For Example:

1. **interface** I{}
2. **class** A{}
3. **class** B **extends** A **implements** I{}

EXAMPLE OF RUNTIME POLYMORPHISM

1. **class** Bike{
2. **void** run(){System.out.println("running");}
3. }
4. **class** Splendor **extends** Bike{
5. **void** run(){System.out.println("running safely with 60k m");}
- 6.
7. **public static void** main(String args[]){
8. bike = **new** Splendor(); //upcasting
9. b.run();
10. }
11. }

Output:

running safely with 60km.

1. **class** Bank{
2. **float** getRateOfInterest(){**return** 0;}
3. }
4. **class** SBI **extends** Bank{
5. **float** getRateOfInterest(){**return** 8.4f;}
6. }
7. **class** ICICI **extends** Bank{
8. **float** getRateOfInterest(){**return** 7.3f;}}

```
9. }
10. class AXIS extends Bank{
11.     float getRateOfInterest(){return 9.7f;}
12. }
13. class TestPolymorphism{
14.     public static void main(String args[]){
15.         Bank b;
16.         b=new SBI();
17.         System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
18.         b=new ICICI();
19.         System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
20.         b=new AXIS();
21.         System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
22.     }
23. }
24. Output:
```

```
SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7
```

REFERENCE LINKS:

<https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/>

<https://www.javatpoint.com/runtime-polymorphism-in-java>

<https://www.geeksforgeeks.org/overloading-in-java/>

[Packages in java:](#)

Package in [Java](#) is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

- Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

Advantage of Java Package

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collision.

How packages work?

Package names and directory structure are closely related. For example if a package name is `college.staff.cse`, then there are three directories, `college`, `staff` and `cse` such that `cse` is present in `staff` and `staff` is present `college`. Also, the directory `college` is accessible through [CLASSPATH](#) variable, i.e., path of parent directory of `college` is present in `CLASSPATH`. The idea is to make sure that classes are easy to locate.

Package naming conventions : For example, in a college, the recommended convention is `college.tech.cse`, `college.tech.ee`, `college.art.history`, etc.

Adding a class to a Package : We can add more classes to a created package by using package name at the top of the program and saving it in the package directory.

Subpackages: Packages that are inside another package are the **subpackages**. These are not imported by default, they have to be imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default

access specifiers.

Example :

```
import java.util.*;
```

util is a subpackage created inside **java** package.

Accessing classes inside a package

Consider following two statements :

```
// import the Vector class from util package.  
import java.util.Vector;  
  
// import all the classes from util package  
import java.util.*;
```

- First Statement is used to import **Vector** class from **util** package which is contained inside **java**.
- Second statement imports all the classes from **util** package.

```
// Java program to demonstrate accessing of members when  
// corresponding classes are imported and not imported.  
  
import java.util.Vector;  
  
  
public class ImportDemo  
{  
    public ImportDemo()  
    {  
        // java.util.Vector is imported, hence we are  
        // able to access directly in our code.  
        Vector newVector = new Vector();  
  
        // java.util.ArrayList is not imported, hence  
        // we were referring to it using the complete
```

```

// package.

java.util.ArrayList newList = new java.util.ArrayList();

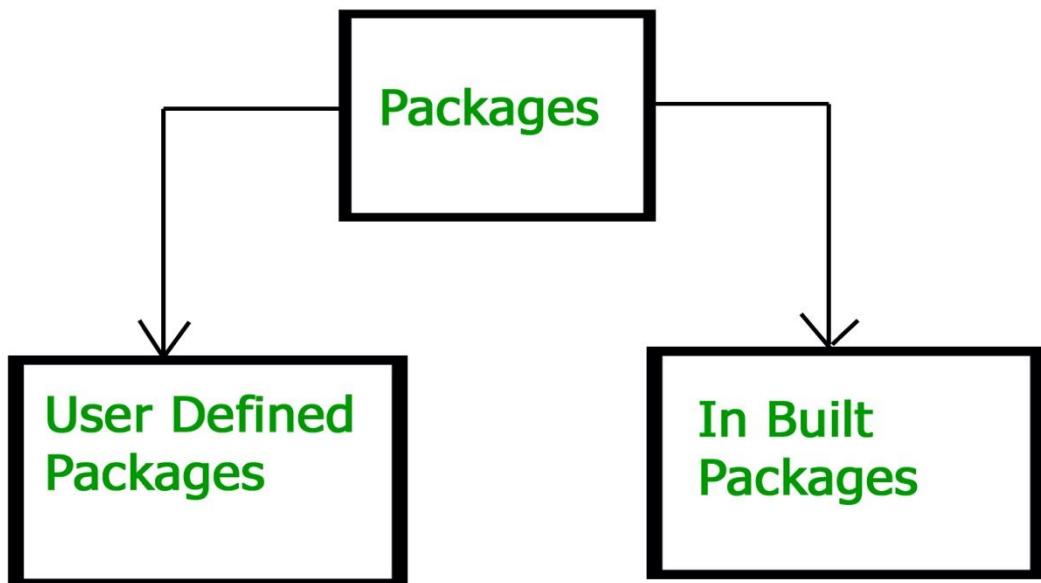
}

public static void main(String arg[])
{
    new ImportDemo();
}

}

```

Types of packages:



1. Built-in Packages

These packages consist of a large number of classes which are a part of Java **API**. Some of the commonly used built-in packages are:

- 1) **java.lang**: Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
- 2) **java.io**: Contains classed for supporting input / output operations.
- 3) **java.util**: Contains utility classes which implement data structures like Linked List,

Dictionary and support ; for Date / Time operations.

- 4) **java.applet**: Contains classes for creating Applets.
- 5) **java.awt**: Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).
- 6) **java.net**: Contain classes for supporting networking operations.

2. User-defined packages

These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names**.

```
3. // Name of the package must be same as the directory
4. // under which this file is saved
5. package myPackage;
6.
7. public class MyClass
8. {
9.     public void getNames(String s)
10.    {
11.        System.out.println(s);
12.    }
13. }
```

14. Now we can use the **MyClass** class in our program.

```
/* import 'MyClass' class from 'names' myPackage */
import myPackage.MyClass;
```

```
public class PrintName
{
    public static void main(String args[])
    {
        // Initializing the String variable
        // with a value
        String name = "GeeksforGeeks";

        // Creating an instance of class MyClass in
        // the package.
        MyClass obj = new MyClass();
```

```
    obj.getNames(name);  
}  
}
```

Note : **MyClass.java** must be saved inside the **myPackage** directory since it is a part of the package.

```
// Note static keyword after import.  
  
import static java.lang.System.*;  
  
  
class StaticImportDemo  
{  
  
    public static void main(String args[])  
    {  
  
        // We don't need to use 'System.out'  
        // as imported using static.  
  
        out.println("Thejesh");  
    }  
}
```

Output:

```
Thejesh
```

Handling name conflicts

The only time we need to pay attention to packages is when we have a name conflict . For example both, java.util and java.sql packages have a class named Date. So if we import both packages in program as follows:

```
import java.util.*;  
  
import java.sql.*;  
  
  
//And then use Date class, then we will get a compile-time error :  
  
  
Date today ; //ERROR-- java.util.Date or java.sql.Date?
```

The compiler will not be able to figure out which Date class do we want. This problem can be solved by using a specific import statement:

```
import java.util.Date;  
import java.sql.*;
```

If we need both Date classes then, we need to use a full package name every time we declare a new object of that class.

For Example:

```
java.util.Date deadLine = new java.util.Date();  
java.sql.Date today = new java.sql.Date();
```

Syntax for compiling a package:

```
javac -d directory javaFileName
```

Important points:

1. Every class is part of some package.
2. If no package is specified, the classes in the file goes into a special unnamed package (the same unnamed package for all files).
3. Java and javax are the root packages.
4. If we have all package, import and class to be used in a file then the hierarchy is like first declare package, then import and then class.

Eg:

```
1. package mypack;  
2. import pack.*;  
3.  
4. class B{  
5.     public static void main(String args[]){  
6.         A obj = new A();  
7.         obj.msg();  
8.     }  
9. }
```

Access Specifiers (or) Access Modifiers in Java

As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor , variable , method or data member. There are four types of access modifiers available in java:

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

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

1.Default: When no access modifier is specified for a class , method or data member – It is said to be having the **default** access modifier by default.

The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible **only within the same package**.

In this example, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of second package.

```
//Java program to illustrate default modifier  
package p1;
```

```

//Class Geeks is having Default access modifier

class Geek

{
    void display()
    {
        System.out.println("Hello World!");
    }
}

//Java program to illustrate error while
//using class from different package with
//default modifier

package p2;
import p1.*;

//This class is having default access modifier

class GeekNew
{
    public static void main(String args[])
    {
        //accessing class Geek from package p1
        Geeks obj = new Geeks();

        obj.display();
    }
}

```

Output:

Compile time error

2.Private: The private access modifier is specified using the keyword **private**.

- The methods or data members declared as private are accessible only **within the class** in which they are declared.

- Any other **class of same package will not be able to access** these members.
- Top level Classes or interface can not be declared as private because
 1. private means “only visible within the enclosing class”.
 2. protected means “only visible within the enclosing class and any subclasses”
 Classes cannot have private or protected access specifiers whereas subclasses can have private and protected access specifiers.

In this example, we will create two classes A and B within same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

```
//Java program to illustrate error while
//using class from different package with
//private modifier
package p1;

class A
{
    private void display()
    {
        System.out.println("GeeksforGeeks");
    }
}

class B
{
    public static void main(String args[])
    {
        A obj = new A();
        //trying to access private method of another class
        obj.display();
    }
}
```

Output:

```
error: display() has private access in A
        obj.display();
```

3.protected: The protected access modifier is specified using the keyword **protected**.

- The methods or data members declared as protected are **accessible within same package or sub classes in different package**.

In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.

```
//Java program to illustrate
//protected modifier

package p1;

//Class A
public class A
{
    protected void display()
    {
        System.out.println("GeeksforGeeks");
    }
}

//Java program to illustrate
//protected modifier

package p2;

import p1.*; //importing all classes in package p1

//Class B is subclass of A
class B extends A
{
    public static void main(String args[])
    {
        B obj = new B();
        obj.display();
    }
}
```

```
}
```

```
}
```

Output:

GeeksforGeeks

4. public: The public access modifier is specified using the keyword **public**.

- The public access modifier has the **widest scope** among all other access modifiers.
- Classes, methods or data members which are declared as public are **accessible from every where** in the program. There is no restriction on the scope of a public data members.

```
//Java program to illustrate  
//public modifier  
  
package p1;  
  
public class A  
  
{  
  
    public void display()  
  
    {  
  
        System.out.println("GeeksforGeeks");  
  
    }  
  
}  
  
package p2;  
  
import p1.*;  
  
class B  
  
{  
  
    public static void main(String args[])  
  
    {  
  
        A obj = new A;  
  
        obj.display();  
  
    }  
  
}
```

Output:

GeeksforGeeks

Ascending order of access specifiers is as follows

private, default, protected and public

External references

1. <https://www.javatpoint.com/package>
2. <https://www.geeksforgeeks.org/packages-in-java/>
3. <https://www.geeksforgeeks.org/access-modifiers-java/>

POLYMORPHISM

Polymorphism in Java

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

1. Compile time polymorphism

=>method overloading:

Discussed before

=> Operator Overloading:

Java also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands, adds them and when placed between string operands, concatenates them.

In java, Only “+” operator can be overloaded:

To add integers => "5 + 4 = 9"

To concatenate strings=> "Thejesh" + " k" = "Thejesh K"

*But if you add two char values it returns an integer value => 'c' + 'j' = 205(ascii value addition).

// Java program for Operator overloading

```
class OperatorOVERDDN {
```

```
    void operator(String str1, String str2)
```

```
{
```

```
String s = str1 + str2;  
System.out.println("Concatinated String - "  
+ s);  
}  
  
void operator(int a, int b)  
{  
    int c = a + b;  
    System.out.println("Sum = " + c);  
}  
/*void operator(char a, char b)  
{  
    String c = a + b ;  
    System.out.println("Sum = " + c);  
} */  
void operator(char a, char b, String str)  
{  
    String c = a + b + str;  
    System.out.println("Sum = " + c);  
}  
}  
  
class Main {  
    public static void main(String[] args)
```

```
{  
    OperatorOVERDDN obj = new  
    OperatorOVERDDN();  
    obj.operator(2, 3);  
    obj.operator('c','j');  
    obj.operator('c','j',"string");  
    obj.operator("joe", "now");  
}  
}
```

Output:

```
-----  
Sum = 5  
Sum = 205  
Sum = 205string  
Concatinated String - joenow
```

2.Runtime polymorphism

```
-----
```

Already discussed.

Queue Interface In Java

The Queue interface is available in `java.util` package and extends the Collection interface. The queue collection is used to hold the elements about to be processed and provides various operations like the insertion, removal etc. It is an ordered list of objects with its use limited to insert elements at the end of the list and deleting elements from the start of list i.e. it follows the FIFO or the First-In-First-Out principle. Being an interface the queue needs a concrete class for the declaration and the most common classes are the [PriorityQueue](#) and [LinkedList](#) in Java. It is to be noted that both the implementations are not thread safe. [PriorityBlockingQueue](#) is one alternative implementation if thread safe implementation is needed. Few important characteristics of Queue are:

- The Queue is used to insert elements at the end of the queue and removes from the beginning of the queue. It follows FIFO concept.
- The Java Queue supports all methods of Collection interface including insertion, deletion etc.
- [LinkedList](#), [ArrayBlockingQueue](#) and [PriorityQueue](#) are the most frequently used implementations.
- If any null operation is performed on BlockingQueues, `NullPointerException` is thrown.
- BlockingQueues have thread-safe implementations.
- The Queues which are available in `java.util` package are Unbounded Queues
- The Queues which are available in `java.util.concurrent` package are the Bounded Queues.
- BlockingQueues are used to implement Producer/Consumer based applications.
- All Queues except the Deques supports insertion and removal at the tail and head of the queue respectively. The Deques support element insertion and removal at both ends.

REFERENCE LINK:

<https://www.geeksforgeeks.org/queue-interface-java/>

Java Queue Methods

1. **int size():** to get the number of elements in the Set.
2. **boolean isEmpty():** to check if Set is empty or not.

3. **boolean contains(Object o):** Returns true if this Set contains the specified element.
4. **Iterator iterator():** Returns an iterator over the elements in this set. The elements are returned in no particular order.
5. **boolean removeAll(Collection c):** Removes from this set all of its elements that are contained in the specified collection (optional operation).
6. **boolean retainAll(Collection c):** Retains only the elements in this set that are contained in the specified collection (optional operation).
7. **void clear():** Removes all the elements from the set.
8. **E remove():** Retrieves and removes the head of this queue.
9. **E poll():** Retrieves and removes the head of this queue, or returns null if this queue is empty.
10. **E peek():** Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
11. **boolean offer(E e):** Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
12. **E element():** Retrieves, but does not remove, the head of this queue.
13. **boolean add(E e):** Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.
14. **Object[] toArray():** Returns an array containing all of the elements in this set. If this set makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.

```
import java.util.*;  
  
public class QueueExample {  
  
    public static void main(String[] args) {  
  
        Queue<String> queue = new LinkedList<>();  
  
        queue.add("one");  
  
        queue.add("two");  
  
        queue.add("three");  
  
        queue.add("four");  
    }  
}
```

```
System.out.println(queue);
queue.remove("three");
System.out.println(queue);
System.out.println("Queue Size: " + queue.size());
System.out.println("Queue Contains element 'two' or not? : " +
queue.contains("two"));
// To empty the queue
queue.clear();
}}
```

Output:

[one, two, three, four]

[one, two, four]

Queue Size: 3

Queue Contains element 'two' or not? : true

Java Queue Common Operations

Java Queue supports all operations supported by Collection interface and some more operations. It supports almost all operations in two forms.

- One set of operations throws an exception if the operation fails.
- The other set of operations returns a special value if the operation fails.

The following table explains all Queue common operations briefly.

Operation	Throws exception	Special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

- **Queue.add(e):**

It throws an exception if the operation fails.

- **Queue.offer(e):**

It returns a special value if the operation fails.

Queue add() operation

The add() operation is used to insert new element into the queue. If it performs insert operation successfully, it returns “true” value. Otherwise it throws `java.lang.IllegalStateException`.

Let us develop one simple example to demonstrate this functionality.

```
import java.util.concurrent.*;
public class QueueAddOperation {
    public static void main(String[] args) {
        BlockingQueue<String> queue = new ArrayBlockingQueue<>(2);
```

```
        System.out.println(queue.add("one"));
        System.out.println(queue.add("two"));
        System.out.println(queue);
        System.out.println(queue.add("three"));
        System.out.println(queue);
    }
}
```

Output:-

When we run above program, We will get the following output:

true

true

[one, two]

Exception in thread "main" java.lang.IllegalStateException: Queue full

As our queue is limited to two elements, when we try to add third element using BlockingQueue.add(), it throws an exception as shown above.

Queue offer() operation

The offer() operation is used to insert new element into the queue. If it performs insert operation successfully, it returns “true” value. Otherwise it returns “false” value.

Let us develop one simple example to demonstrate this functionality.

```
import java.util.concurrent.*;
public class QueueOfferOperation {
    public static void main(String[] args) {
        BlockingQueue<String> queue = new ArrayBlockingQueue<>(2);
        System.out.println(queue.offer("one"));
        System.out.println(queue.offer("two"));
        System.out.println(queue);
        System.out.println(queue.offer("three"));
```

```
        System.out.println(queue);
    }
}
```

Output:-

When we run above program, We will get the following output:

true

true

[one, two]

false

[one, two]

As our queue is limited to two elements, when we try to add third element using BlockingQueue.offer() operation, it returns “false” value as shown above.

Java Queue Delete Operations

In this section, we will discuss about Java Queue Delete operation in-detail with some useful examples. The Delete operations returns the head element of the queue, if it performs successfully. As we know, Queue supports delete operation in two forms:

Queue.remove():

It throws an exception if the operation fails.

Queue.poll():

It returns a special value if the operation fails.

NOTE:- Here special value may be either “false” or “null”

Queue remove() operation

The remove() operation is used to delete an element from the head of the queue. If it performs delete operation successfully, it returns the head element of the queue. Otherwise it throws java.util.NoSuchElementException.

Let us develop one simple example to demonstrate this functionality.

```
import java.util.*;
```

```
public class QueueRemoveOperation
{
    public static void main(String[] args)
    {
        Queue<String> queue = new LinkedList<>();
        queue.offer("one");
        queue.offer("two");
        System.out.println(queue);
        System.out.println(queue.remove());
        System.out.println(queue.remove());
        System.out.println(queue.remove());
    }
}
```

Output:-

When we run above program, We will get the following output:

[one, two]

one

two

Exception in thread "main" java.util.NoSuchElementException

As our queue has only two elements, when we try to call remove() method for third time, it throws an exception as shown above.

NOTE:-

Queue.remove(element) is used to delete a specified element from the queue. If it performs delete operation successfully, it returns “true” value. Otherwise it returns “false” value.

Queue poll() operation

The poll() operation is used to delete an element from the head of the queue. If it performs delete operation successfully, it returns the head element of the queue. Otherwise it returns “null” value.

Let us develop one simple example to demonstrate this functionality.

```
import java.util.*;
public class QueuePollOperation
{
    public static void main(String[] args)
    {
        Queue<String> queue = new LinkedList<>();
        queue.offer("one");
        queue.offer("two");
        System.out.println(queue);
        System.out.println(queue.poll());
        System.out.println(queue.poll());
        System.out.println(queue.poll());
    }
}
```

Output:-

When we run above program, We will get the following output:

```
[one, two]
one
two
null
```

As our queue has only two elements, when we try to call poll() method for third time, it returns null value as shown above.

Java Queue Examine Operations

In this section, we will discuss about Java Queue Examine operations in-detail with some useful examples. If this operation performs successfully, it returns the head element of the queue without removing it. As we know, Queue supports examine operation in two forms:

Queue.element():

It throws an exception if the operation fails.

Queue.peek():

It returns a special value if the operation fails.

NOTE:- Here special value may be either “false” or “null”

Queue element() operation

The element() operation is used to retrieve an element from the head of the queue, without removing it. If it performs examine operation successfully, it returns the head element of the queue. Otherwise it throws java.util.NoSuchElementException.

Let us develop one simple example to demonstrate this functionality.

```
import java.util.*;  
  
public class QueueElementOperation {  
    public static void main(String[] args) {  
        Queue<String> queue = new LinkedList<>();  
        queue.add("one");  
        System.out.println(queue.element());  
        System.out.println(queue);  
        queue.clear();  
        System.out.println(queue.element());  
    }  
}
```

Output:-

When we run above program, We will get the following output:

one

[one]

Exception in thread "main" java.util.NoSuchElementException

If we try to call element() method on empty Queue, it throws an exception as shown above

Queue peek() operation

The peek() operation is used to retrieve an element from the head of the queue, without removing it. If it performs examine operation successfully, it returns the head element of the queue. Otherwise it returns null value.

Let us develop one simple example to demonstrate this functionality.

```
import java.util.*;  
  
public class QueuePeekOperation {  
    public static void main(String[] args) {  
        Queue<String> queue = new LinkedList<>();  
        queue.add("one");  
        System.out.println(queue.peek());  
        System.out.println(queue);  
        queue.clear();  
        System.out.println(queue.peek());  
    }  
}
```

Output:-

When we run above program, We will get the following output:

one

[one]

null

If we try to call peek() method on empty Queue, it returns null value, but does NOT throw an exception as shown above.

REFERENCE LINK:

<https://www.journaldev.com/13244/java-queue>

Java Queue Categories

In Java, we can find many Queue implementations. We can broadly categorize them into the following two types

- Blocking Queues
- Non-Blocking Queues
 - All Queues which implement BlockingQueue interface are BlockingQueues and rest are Non-Blocking Queues.
 - BlockingQueues blocks until it finishes its job or time out, but Non-BlockingQueues do not.
 - Some Queues are Deques and some queues are PriorityQueues.

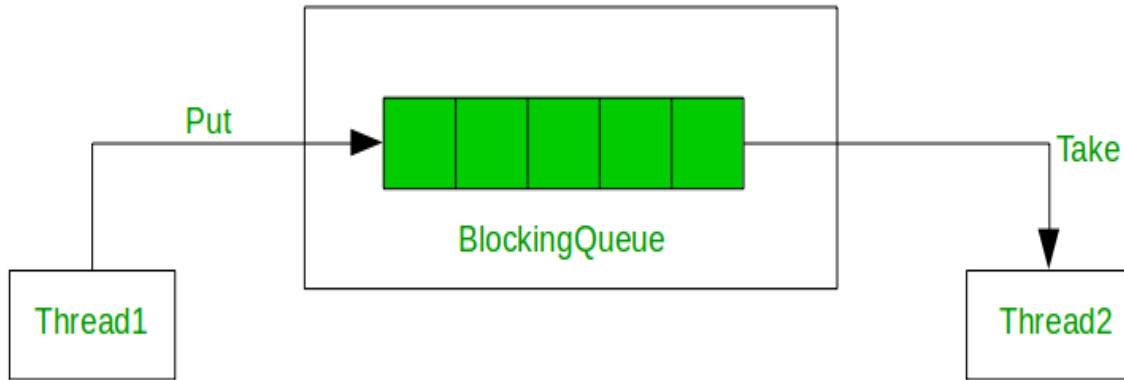
BlockingQueue Interface in Java

BlockingQueue interface in Java is added in Java 1.5 along with various other concurrent Utility classes like ConcurrentHashMap, Counting Semaphore, CopyOnWriteArrayList etc. BlockingQueue interface supports flow control (in addition to queue) by introducing blocking if either BlockingQueue is full or empty. A thread trying to enqueue an element in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more element or clearing the queue completely. Similarly it blocks a thread trying to delete from an empty queue until some other threads inserts an item.

BlockingQueue does not accept null value. If we try to enqueue null item, then it throws NullPointerException.

Java provides several BlockingQueue implementations such as LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue, SynchronousQueue etc.

Java BlockingQueue interface implementations are thread-safe. Java 5 comes with BlockingQueue implementations in the **java.util.concurrent package**.



BlockingQueue Types

The BlockingQueue are two types-

- **Unbounded Queue:** The Capacity of blocking queue will be set to Integer.MAX_VALUE. In case of unbounded blocking queue, queue will never block because it could grow to a very large size. when you add elements it's size grow.

Syntax:

```
BlockingQueue blockingQueue = new LinkedBlockingDeque();
```

- **Bounded Queue:** The second type of queue is the bounded queue. In case of bounded queue you can create a queue by passing the capacity of queue in queues constructor:

Syntax:

```
// Creates a Blocking Queue with capacity 5
```

```
BlockingQueue blockingQueue = new LinkedBlockingDeque(5);
```

Bounded Queues are queues which are bounded by capacity that means we need to provide the max size of the queue at the time of creation. For example ArrayBlockingQueue (see previous example).

Unbounded Queues are queues which are NOT bounded by capacity that means we should not provide the size of the queue. For example LinkedList (see previous example).

All Queues which are available in `java.util` package are Unbounded Queues and Queues which are available in `java.util.concurrent` package are Bounded Queues.

Methods in Blocking Queue Interface

MODIFIER AND TYPE	METHOD SYNTAX	USED FOR	DESCRIPTION
	boolean add(E e)	Insertion	<p>Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an <code>IllegalStateException</code> if no space is currently available.</p>

			Returns true if this queue contains the specified element.
boolean	contains(Object o)	Examine	
			Retrieving elements from this queue and adds them to the given collection.
int	drainTo(Collection c)	Removal	
			Removes at most the given number of available elements from this queue and adds them to the given collection.
int	drainTo(Collection c, int maxElements)	Removal	
			Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning
boolean	offer(E e)	Insertion	

			true upon success and false if no space is currently available.
			Inserts the specified element into this queue, waiting up to the specified wait time if necessary for space to become available.
boolean	offer(E e, long timeout, TimeUnit unit)	Insertion	Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available.
E	poll(long timeout, TimeUnit unit)	Retrieving or Removal	Inserts the specified element into this queue, waiting if necessary for
void	put(E e)	Insertion	

space to become
available.

Returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking, or `Integer.MAX_VALUE` if there is no intrinsic

int remainingCapacity() Examine limit.

Removes a single instance of the specified element from this queue,

`boolean remove(Object o)`+ Removal if it is present.

E	take()	Retrieving or Removal	Retrieves and removes the head of this queue, waiting if necessary until
---	--------	-----------------------	--

an element becomes
available.

```
//Example of queue.offer()
1. java.util.concurrent.ArrayBlockingQueue;
2. import java.util.concurrent.TimeUnit;
3. public class ArrayBlockingQueueOfferExample2 {
4.     public static void main(String[] args) throws InterruptedException {
5.         int capacity =5;
6.         ArrayBlockingQueue<String> queue = new ArrayBlockingQueue<String>(ca
    pacity);
7.         queue.add("Reema");
8.         queue.add("Rahul");
9.         queue.add("Rita");
10.        queue.add("Ramesh");
11.        //Inserts the element at the tail of queue,waiting up to the specified
    waiting time.
12.        queue.offer("Gita",67, TimeUnit.MILLISECONDS);
13.        for(String xyz:queue){
14.            System.out.println(xyz);
15.        }
16.    }
17. }
```

Output:

Reema
Rahul
Rita
Ramesh

Java Program Demonstrate drainTo(Collection c)
// method of BlockingQueue.

```
import java.util.ArrayList;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class GFG {
```

```

// create a Employee Object with
// position and salary as an attribute
public class Employee {

    public String name;
    public String position;
    public String salary;
    Employee(String name, String position, String salary)
    {
        this.name = name;
        this.position = position;
        this.salary = salary;
    }
    @Override
    public String toString()
    {
        return "Employee [name=" + name + ", position="
            + position + ", salary=" + salary + "]";
    }
}

// Main Method
public static void main(String[] args)
{
    GFG gfg = new GFG();
    gfg.containsMethodExample();
}

public void containsMethodExample()
{
    // define capacity of BlockingQueue
    int capacity = 50;

    // create object of BlockingQueue
    BlockingQueue<Employee> BQ
        = new LinkedBlockingQueue<Employee>(capacity);

    // create a ArrayList to pass as parameter to drainTo()
    ArrayList<Employee> collection = new ArrayList<Employee>();

    // add Employee object to queue
    Employee emp1 = new Employee("Aman", "Analyst", "24000");
}

```

```

Employee emp2 = new Employee("Sachin", "Developer", "39000");
BQ.add(emp1);
BQ.add(emp2);

// printing ArrayList and queue
System.out.println("Before drainTo():");
System.out.println("BlockingQueue : \n"
+ BQ.toString());
System.out.println("ArrayList : \n"
+ collection);

// Apply drainTo method and pass collection as parameter
int response = BQ.drainTo(collection);

// print no of element passed
System.out.println("\nNo of element passed: " + response);

// printing ArrayList and queue after applying drainTo() method
System.out.println("\nAfter drainTo():");
System.out.println("BlockingQueue : \n"
+ BQ.toString());
System.out.println("ArrayList : \n"
+ collection);
}
}

```

Output:

Before drainTo():

LinkedBlockingQueue :

[Employee [name=Aman, position=Analyst, salary=24000], Employee [name=Sachin, position=Developer, salary=39000]]

ArrayList :

[]

No of element passed: 2

After drainTo():

LinkedBlockingQueue :

[]

ArrayList :

```
[Employee [name=Aman, position=Analyst, salary=24000], Employee  
[name=Sachin, position=Deve
```

```
// Example of queue.remainingCapacity()  
X  
1. import java.util.concurrent.ArrayBlockingQueue;  
2. import java.util.concurrent.BlockingQueue;  
3. public class ArrayBlockingQueueRemainingCapacityExample1 {  
4.     public static void main(String[] args) {  
5.         int capacity = 10;  
6.         BlockingQueue<Integer> queue = new ArrayBlockingQueue<Integer>(capa-  
city);  
7.         for (int i = 1; i <= 6; i++) {  
8.             // Adding items add() to the tail of queue  
9.             queue.add(i);  
10.        }  
11.        System.out.println("Queue : "+queue);  
12.        int val =queue.remainingCapacity();  
13.        System.out.println("Remaining capacity of the queue = "+val);  
14.    }  
15. }
```

Output:

```
Queue : [1, 2, 3, 4, 5, 6]  
Remaining capacity of the queue = 4
```

SCOPE

Identify the scope of variables

Scope is the region where a variable can be accessed

-->class level scope

variables can be accessed within a class(for all methods)

-->Local scope(Method scope)

scope will be valid within method

-->block scope

scope will be valid within block of code within{}

if class variable and method variable have same names then "this keyword" is used to differentiate class n method variables

```
public class Test
{
    static int x = 11;
    private int y = 33;
    public void method1(int x)
    {
        Test t = new Test();
        this.x = 22;
        y = 44;

        System.out.println("Test.x: " + Test.x); //22
        System.out.println("t.x: " + t.x); //22
        System.out.println("t.y: " + t.y); //33
        System.out.println("y: " + y); //44
    }
}
```

```
public static void main(String args[])
{
    Test t = new Test();
    t.method1(5);
}

}

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

        for (int x = 0; x < 4; x++)
        {
            System.out.println(x);
        }

        //int x=10;

        // Will produce error
        System.out.println(x);
    }
}
```

LINKS

<https://www.inf.unibz.it/~calvanese/teaching/05-06-ip/lecture-notes/uni03/node15.html>

<https://www.geeksforgeeks.org/variable-scope-in-java/>

Use local variable type inference

>Here we use "var" to declare and initialize the variables, it can be used to identify the type of a variable.

>The type of a variable can be determined by the value assigned to it(R.H.S)

>After compilation, the var(to int, String, ...) can be modified to exact type of a variable based on value assigned.

>it does not allow "var" without initialization(var x).

>But it's not a dynamically typed language

>A language is dynamically-typed if the type of a variable is checked during run-time.

>"var" is converted to data-type after compilation.

>var can be name of variable, method or package.

>var is not a keyword.

```
var var = 5; // syntactically correct
```

```
// var is the name of the variable
```

```
public static void var() { // syntactically correct
```

```
}
```

```
package var; // syntactically correct
```

LINKS

<https://www.journaldev.com/19871/java-10-local-variable-type-inference>

Set collection in java

Set is an interface which extends Collection. It is an unordered collection of objects in which duplicate values cannot be stored.

Basically, Set is implemented by HashSet, LinkedHashSet or TreeSet (sorted representation).

Set has various methods to add, remove clear, size, etc to enhance the usage of this interface

```
•    // Java code for adding elements in Set
•    import java.util.*;
•    public class Set_example
•    {
•        public static void main(String[] args)
•        {
•            // Set deonstration using HashSet
•            Set<String> hash_Set = new HashSet<String>();
•            hash_Set.add("Geeks");
•            hash_Set.add("For");
•            hash_Set.add("Geeks");
•            hash_Set.add("Example");
•            hash_Set.add("Set");
•            System.out.print("Set output without the duplicates");
•
•            System.out.println(hash_Set);
•
•            // Set deonstration using TreeSet
•            System.out.print("Sorted Set after passing into TreeSet");
•            Set<String> tree_Set = new TreeSet<String>(hash_Set);
•            System.out.println(tree_Set);
•        }
•    }
```

Note:- we have entered a duplicate entity but it is not displayed in the output. Also, we can directly sort the entries by passing the unordered Set in as the parameter of TreeSet.

Output:

```
Set output without the duplicates[Set, Example, Geeks, for]
```

```
Sorted Set after passing into TreeSet[Example, For, Geeks, Set]
```

Note: As we can see the duplicate entry “Geeks” is ignored in the final output, Set interface doesn’t allow duplicate entries.

Now we will see some of the basic operations on the Set i.e. Union, Intersection and Difference.

Let’s take an example of two integer Sets:

- [1, 3, 2, 4, 8, 9, 0]
- [1, 3, 7, 5, 4, 0, 7, 5]

Union

In this, we could simply add one Set with other. Since the Set will itself not allow any duplicate entries, we need not take care of the common values.

Expected Output:

```
Union : [0, 1, 2, 3, 4, 5, 7, 8, 9]
```

Intersection

We just need to retain the common values from both Sets.

Expected Output:

```
Intersection : [0, 1, 3, 4]
```

Difference

We just need to remove all the values of one Set from the other.

Expected Output:

```
Difference : [2, 8, 9]
```

```
// Java code for demonstrating union, intersection and difference
```

```

// on Set

import java.util.*;

public class Set_example
{
    public static void main(String args[])
    {
        Set<Integer> a = new HashSet<Integer>();
        a.addAll(Arrays.asList(new Integer[] {1, 3, 2, 4, 8, 9, 0}));
        Set<Integer> b = new HashSet<Integer>();
        b.addAll(Arrays.asList(new Integer[] {1, 3, 7, 5, 4, 0, 7, 5}));

        // To find union
        Set<Integer> union = new HashSet<Integer>(a);
        union.addAll(b);
        System.out.print("Union of the two Set");
        System.out.println(union);

        // To find intersection
        Set<Integer> intersection = new HashSet<Integer>(a);
        intersection.retainAll(b);
        System.out.print("Intersection of the two Set");
        System.out.println(intersection);

        // To find the symmetric difference
        Set<Integer> difference = new HashSet<Integer>(a);
        difference.removeAll(b);
        System.out.print("Difference of the two Set");
        System.out.println(difference);
    }
}

```

Output:

Union of the two Set[0, 1, 2, 3, 4, 5, 7, 8, 9]

Intersection of the two Set[0, 1, 3, 4]

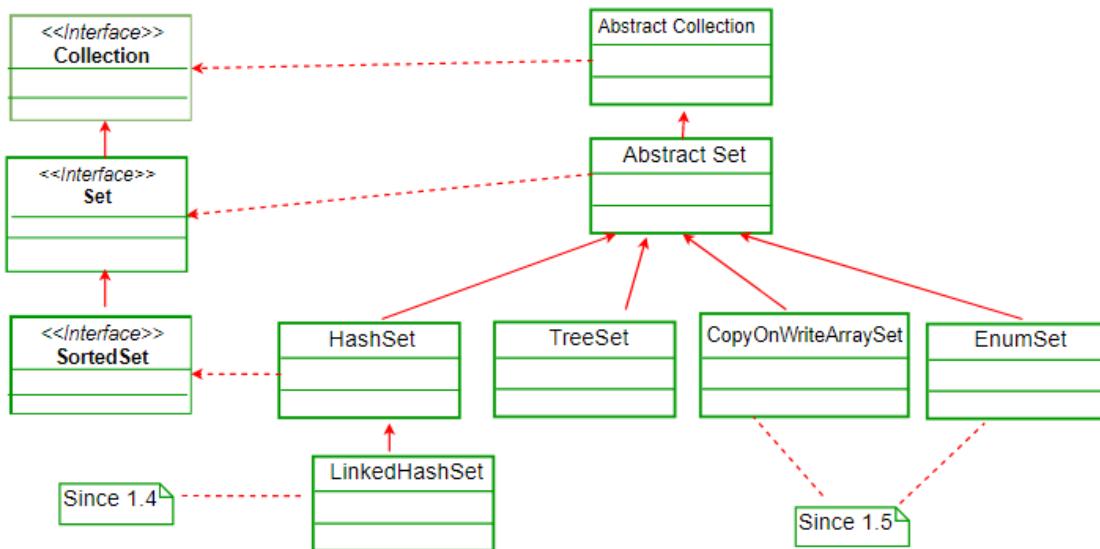
Difference of the two Set[2, 8, 9]

HashSet in Java

The HashSet class implements the Set interface, backed by a hash table which is actually a HashMap instance. No guarantee is made as to the iteration order of the set which means that the class does not guarantee the constant order of elements over time. This class permits the null element. The class also offers constant time performance for the basic operations like add, remove, contains and size assuming the hash function disperses the elements properly among the buckets, which we shall see further in the article.

Few important features of HashSet are:

- Implements [Set Interface](#).
- Underlying data structure for HashSet is hashtable.
- As it implements the Set Interface, duplicate values are not allowed.
- Objects that you insert in HashSet are not guaranteed to be inserted in same order. Objects are inserted based on their hash code.
- NULL elements are allowed in HashSet.
- HashSet also implements Serializable and Cloneable interfaces.



Now for the maintenance of constant time performance, iterating over HashSet requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the

“capacity” of the backing HashMap instance (the number of buckets). Thus, it’s very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

Initial Capacity: The initial capacity means the number of buckets when hashtable (HashSet internally uses hashtable data structure) is created. The number of buckets will be automatically increased if the current size gets full.

Load Factor: The load factor is a measure of how full the HashSet is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

$$\text{load factor} = \frac{\text{Number of stored elements in the table}}{\text{Size of the hash table}}$$

Example: If internal capacity is 16 and load factor is 0.75 then, number of buckets will automatically get increased when the table has 12 elements in it.

NOTE: The implementation in a HashSet is not synchronized, in the sense that if multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be “wrapped” using the Collections.synchronizedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the set as shown below:

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

Constructors in HashSet:

1. **HashSet h = new HashSet();**
Default initial capacity is 16 and default load factor is 0.75.
2. **HashSet h = new HashSet(int initialCapacity);**
default loadFactor of 0.75
3. **HashSet h = new HashSet(int initialCapacity, float loadFactor);**
4. **HashSet h = new HashSet(Collection C);**

```
import java.util.*;
```

```

class Test
{
    public static void main(String[] args)
    {
        HashSet<String> h = new HashSet<String>();

        // Adding elements into HashSet usind add()
        h.add("India");
        h.add("Australia");
        h.add("South Africa");
        h.add("India");// adding duplicate elements

        // Displaying the HashSet
        System.out.println(h);
        System.out.println("List contains India or not:" +
                           h.contains("India"));

        // Removing items from HashSet using remove()
        h.remove("Australia");

        System.out.println("List after removing Australia:" + h);

        // Iterating over hash set items
        System.out.println("Iterating over list:");
        Iterator<String> i = h.iterator();
        while (i.hasNext())
            System.out.println(i.next());
    }
}

```

Output:

[South Africa, Australia, India]

```
List contains India or not:true  
List after removing Australia:[South Africa, India]  
Iterating over list:  
South Africa  
India
```

Internal working of a HashSet

All the classes of Set interface internally backed up by Map. HashSet uses HashMap for storing its object internally. You must be wondering that to enter a value in HashMap we need a key-value pair, but in HashSet we are passing only one value.

Storage in HashMap

Actually the value we insert in HashSet acts as key to the map Object and for its value java uses a constant variable. So in key-value pair all the values will be same.

Implementation of HashSet in java doc:

```
private transient HashMap map;  
  
// Constructor - 1  
// All the constructors are internally creating HashMap Object.  
public HashSet()  
{  
    // Creating internally backing HashMap object  
    map = new HashMap();  
}  
  
// Constructor - 2  
public HashSet(int initialCapacity)  
{  
    // Creating internally backing HashMap object  
    map = new HashMap(initialCapacity);  
}
```

```
// Dummy value to associate with an Object in Map  
private static final Object PRESENT = new Object();
```

If we look at add() method of HashSet class:

```
public boolean add(E e)  
{  
    return map.put(e, PRESENT) == null;  
}
```

We can notice that, add() method of HashSet class internally calls put() method of backing HashMap object by passing the element you have specified as a key and constant "PRESENT" as its value.

remove() method also works in the same manner. It internally calls remove method of Map interface.

```
public boolean remove(Object o)  
{  
    return map.remove(o) == PRESENT;  
}
```

Methods in HashSet:

1. [boolean add\(E e\)](#): Used to add the specified element if it is not present, if it is present then return false.
2. [void clear\(\)](#): Used to remove all the elements from set.
3. [boolean contains\(Object o\)](#): Used to return true if an element is present in set.
4. [boolean remove\(Object o\)](#): Used to remove the element if it is present in set.
5. [Iterator iterator\(\)](#): Used to return an iterator over the element in the set.
6. [boolean isEmpty\(\)](#): Used to check whether the set is empty or not. Returns true for empty and false for non-empty condition for set.
7. [int size\(\)](#): Used to return the size of the set.
8. [Object clone\(\)](#): Used to create a shallow copy of the set.

STRING BUILDER

When to use which one :

1) If a string is going to remain constant throughout the program, then use String class object because a String object is immutable.

2) If a string can change (example: lots of logic and operations in the construction of the string)

and will only be accessed from a single thread, using a StringBuilder is good enough.

3) If a string can change, and will be accessed from multiple threads, use a

Creating StringBuilder Objects

The StringBuilder class has four overloaded constructors.

StringBuilder()//creates an empty string Builder with the initial capacity of 16.

StringBuilder(CharSequence seq)//

StringBuilder(int capacity)//creates an empty string Builder with the specified capacity as length.

StringBuilder(String str)//creates a string Builder with the specified string.

Let's use these constructors in a sample program.

```
class StringBuilderObjects {
```

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

```
        StringBuilder sb1 = new StringBuilder(); // 1
```

```
        StringBuilder sb2 = new StringBuilder(sb1); // 2
```

```
        StringBuilder sb3 = new StringBuilder(10); // 3
```

```
        StringBuilder sb4 = new StringBuilder("Hi"); // 4
```

```
 }  
 }  
  
-----
```

How StringBuilder capacity increases ?

Line 1 creates StringBuilder object with no characters in it and an initial capacity of 16 characters. After you add 17th character to StringBuilder object , it's capacity will be increased to 34 . The formula used by the StringBuilder to increase its capacity.

```
newCapacity = (intialcapacity * 2) + 2.  
  
-----
```

STRINGBUILDER CLASS METHODS:

```
-----
```

String Builder contains methods. Most of the methods works as like String methods, for example methods such as charAt, indexOf, lastIndexOf, substring and length.

```
public class HelloWorld{  
  
    public static void main(String []args){  
        StringBuilder sb = new StringBuilder();  
  
        StringBuilder sb1 = new StringBuilder();  
        StringBuilder sb2 = new StringBuilder();  
    }  
}
```

```
sb.append(5);
sb.append(false);
sb.append('a');
sb.append(10.5f);
sb.append(10.5);
sb.append("Hi");
System.out.println(sb);
sb1.append(sb);
sb2.append(sb, 1, 4); // appends characters from index 1 to 3 ( because 4 is exclusive).
```

```
System.out.println(sb);
System.out.println(sb1);
System.out.println(sb2);
```

```
char[] name1 = { 'O', 'c', 'a', 'j', 'p' };
char[] name2 = { 'J', 'a', 'v', 'a', '8' };
sb1.append(name1);
System.out.println(sb1);
sb1.append(name2, 1, 3);
System.out.println(sb1); // prints Ocajpava
```

```
//insert()
sb.insert(2, 5); // inserts 5 literal at position 2
sb.insert(4, false); // inserts false literal at position 4
```

```
        sb.insert(6,'a'); // inserts 'a' literal at position 6
        sb.insert(7,10.5f);
        System.out.println(sb);
        System.out.println(sb2);
        sb1.insert(0,sb); // inserts StringBuilder object at position 0
        sb2.insert(1,sb, 1, 4);// inserts characters of sb from index 1 to 3 (because
        4 is exclusive) to sb2 at position 0
        System.out.println(sb);
        System.out.println(sb1);
        System.out.println(sb2);

/* sb.insert(2,name1);
   System.out.println(sb);
   sb.insert(4,name2, 1, 3);
   System.out.println(sb); */

    }
}
```

Output:

```
5falsea10.510.5Hi
5falsea10.510.5Hi
5falsea10.510.5Hi
fal
5falsea10.510.5HiOcajp
```

```
5falsea10.510.5HiOcajpava
5f5afaa10.5lse10.510.5Hi
fal
5f5afaa10.5lse10.510.5Hi
5f5afaa10.5lse10.510.5Hi5falsea10.510.5HiOcajpava
ff5aal
```

```
public class HelloWorld{

    public static void main(String []args){
        //charAt( ), setCharAt( )
        StringBuilder sb = new StringBuilder("Hello");
        char c = sb.charAt(1);
        sb.setCharAt(0, 'N');
        System.out.println(c); // prints e
        System.out.println(sb); // prints Nello

        //reverse(), toString() , replace

        sb.replace(1, 4, "ABC");
        System.out.println(sb); // prints HABC
        StringBuilder sb1 = sb.reverse();
```

```
System.out.println(sb1);

String s = sb.toString();

System.out.println(sb); // prints oCBAH

System.out.println(sb == sb1); // prints true because both sb , sb1
referencing to the same object.

System.out.println(s);// prints oCBAH

System.out.println(sb);

//delete(),deleteCharAt()

sb.delete(2, 4);

System.out.println(sb);

sb.deleteCharAt(0);

System.out.println(sb); // prints eoHi

}

}
```

outPut:

e

Nello

NABC0

oCBAN

oCBAN

true

oCBAN

oCBAN

oCN

CN

```
public class HelloWorld{

    public static void main(String []args){
        StringBuilder sb = new StringBuilder();
        sb.append("OCAJP8");
        System.out.println(sb);
        //capacity()
        System.out.println(sb.capacity());// prints 16 because StringBuilder
        default capacity 16.
        //length()
        System.out.println(sb.length());// prints 6 because we appended 6
        character sequence
        //ensureCapacity()
        // sb.ensureCapacity(10); // It increases the capacity, doesn't changes
        length of character sequence
        System.out.println(sb.capacity()); // prints 34 because capacity increases
        System.out.println(sb.length());// prints 6
        //setLength()
        sb.setLength(8);// It increases the length of character sequence, doesn't
        changes capacity
        System.out.println(sb.capacity()); // prints 34
        System.out.println(sb.length()); // prints 8
    }
}
```

```
//trimToSize()

    sb.trimToSize(); // It decreases the capacity to length of character
sequence, but doesn't changes length of character sequence

    System.out.println(sb.capacity()); // prints 8

    System.out.println(sb.length()); // prints 8


/*
char[] ch = new char[5] ;

sb.getChars(0, 2, ch, 0);

System.out.println(Arrays.toString(ch)); // prints [O,C, , ]

*/
//subString(),instanceOf()

System.out.println(sb);

System.out.println(sb.substring(2)); // prints AJP8

System.out.println(sb.substring(2,5)); // prints AJP

CharSequence c =    sb.subSequence(2, 5);

System.out.println(c); // prints AJP

System.out.println(c instanceof String); // prints true

System.out.println(c instanceof StringBuilder); // prints false

System.out.println(sb instanceof StringBuilder);

}

}


```

outPut:

OCAJP8

16

6

16

6

16

8

8

8

OCAJP8

AJP8

AJP

AJP

true

false

StringBuffer because StringBuffer is synchronous so you have thread-safety.

CONVERSION BETWEEN TYPES OF STRINGS IN JAVA:

Sometimes there is a need of converting a string object of different classes like String,

StringBuffer, StringBuilder to one-another. Below are some techniques to do the same.

FROM STRING TO STRINGBUFFER AND STRINGBUILDER :

This one is easy. We can directly pass String class object to StringBuffer and StringBuilder class constructors. As String class is immutable in java, so for editing a string,

we can perform same by converting it to StringBuffer or StringBuilder class objects.

```
public class Test
{
    public static void main(String[] args)
    {
        String str = "Sumithra";

        // conversion from String object to StringBuffer
        StringBuffer sbr = new StringBuffer(str);
        sbr.reverse();
        System.out.println(sbr);

        // conversion from String object to StringBuilder
        StringBuilder sbl = new StringBuilder(str);
        sbl.append("Gowthamkumar");
        System.out.println(sbl);
    }
}
```

OutPut:

arhtimus

sumithragothamkumar

FROM STRINGBUFFER AND STRINGBUILDER TO STRING :

This conversions can be perform using `toString()` method which is overridden in both `StringBuffer` and `StringBuilder` classes.

Below is the java program to demonstrate the same. Note that while we use `toString()` method, a new `String` object(in Heap area) is allocated and initialized to character sequence currently represented by `StringBuffer` object,

that means the subsequent changes to the `StringBuffer` object do not affect the contents of the `String` object.

```
public class Test
{
    public static void main(String[] args)
    {
        StringBuffer sbr = new StringBuffer("sumithra");
        StringBuilder sbdr = new StringBuilder("Hello");

        // conversion from StringBuffer object to String
        String str = sbr.toString();
        System.out.println("StringBuffer object to String : ");
        System.out.println(str);
```

```
// conversion from StringBuilder object to String  
String str1 = sbdr.toString();  
System.out.println("StringBuilder object to String : ");  
System.out.println(str1);  
  
// changing StringBuffer object sbr  
// but String object(str) doesn't change  
sbr.append("gowthamkumar");  
System.out.println(sbr);  
System.out.println(str);  
  
}  
}
```

Output:

StringBuffer object to String :

sumithra

StringBuilder object to String :

Hello

sumithragowthamkumar

sumithra

FROM STRINGBUFFER TO STRINGBUILDER OR VICE-VERSA :

This conversion is tricky. There is no direct way to convert the same. In this case, We can use a String class object.

We first convert StringBuffer/StringBuilder object to String using `toString()` method and then from String to StringBuilder/StringBuffer using constructors.

```
public class Test
{
    public static void main(String[] args)
    {
        StringBuffer sbr = new StringBuffer("sumithra");

        // conversion from StringBuffer object to StringBuilder
        String str = sbr.toString();
        StringBuilder sbl = new StringBuilder(str);

        System.out.println(sbl);

    }
}
```

Output:

Sumithra

Conclusion:

Objects of String are immutable, and objects of StringBuffer and StringBuilder are mutable.

StringBuffer and StringBuilder are similar, but StringBuilder is faster and preferred over StringBuffer for single threaded program. If thread safety is needed, then StringBuffer is used.

STRNGSS

CREATE AND MANIPULATE STRINGS

>String is a sequence of characters

>String objects which are not created using new keyword are stored in String constant pool, whereas strings created using "new" are stored in heap memory.

```
String str="Hello"
```

```
String str2=new String("Hii")
```

>SCP does not allow duplicates but Heap area allows duplicate objects.

wout new(SCP)	with new(Heap)
s1=hello	new String("hello")
s2=hii	new String("hii")
s3=hello	new String("hello")

Here s1 and s3
points to same
reference

Here, two objects with
same value can be

created.

Approaches to create string objects

String str=new String("hello")

String str="hello"

>String class objects are immutable(modification of same string str can be pointed to another reference)

str1="hello"

str2="hello" //both points to same reference

str1=str1.concat("hii") //creates one more reference for same object

but str2 will

point to the old reference

ASCII VALUES

A-Z-->65-90

a-z-->97-122

0-9-->48-57

METHODS:

>`toString()`-->used to get reference of an object.

returns class-name@hashcode

Ex:

```
HelloWorld h=new HelloWorld();
```

```
System.out.println(h.toString());
```

>`str.length()`

returns the length of string

Ex:

```
String str="Hello";
```

```
System.out.println(str.length());
```

>`str.concat()`-->concat the string at the end,it accepts only one argument of type string.

return new string object only string length is greater than 0, otherwise it returns same object.

Ex:

```
String str="Hello";
```

```
System.out.println(str.concat("macys")); //Hellomacys
```

```
System.out.println(str); //Hello
```

>str.indexOf()

returns the index of char, if char exists

returns -1, if char does not exist

Ex:

```
System.out.println(str.indexOf("z")); //-1
```

```
System.out.println(str.indexOf("l")); //2
```

>str.charAt()

returns char present in that index

Ex:

```
System.out.println(str.charAt("2")) //2
```

>str.compareTo(str) (ASCII value of first char of first string - ASCII value of first char of second string)

returns 0, if both are equal

returns diff of first mismatch, if both are not equal

Ex:

```
String str="Hello";
```

```
System.out.println(str.compareTo("Hello"));      //0  
System.out.println(str.compareTo("hello"));      //-32
```

>str.compareTolgnoreCase()

Ex:

```
String str="hello";  
System.out.println(str.compareTolgnoreCase("hellofdfdf")); // -5
```

>str.toUpperCase()

>str.toLowerCase()

Ex:

```
String str="hello";  
System.out.println(str.toLowerCase());  
System.out.println(str.toUpperCase());
```

>str.trim()

returns string wout whitespaces before n after the
string

Ex:

```
String str="           hii";
```

```
System.out.println(str.trim()); //hii
```

>str.contains()-->searches the sequence of characters in this string.

return true, if seq exists

return false, if not exists

Ex:

```
String name="what do you know about me";
```

```
System.out.println(name.contains(" gknow")); //false
```

>str.startsWith()-->checks if the string with given prefix

>str.endsWith()-->checks if this string ends with given suffix

returns true,if string ends with given suffix

returns false, if not

Ex:

```
String s1="Honey is sweet";
```

```
System.out.println(s1.endsWith("eT")); //false
```

```
System.out.println(s1.startsWith("Ho")); //true
```

>str.equals()-->compares the content of two strings

return true,if all characters are matched

return false,if not

Ex:

```
String str2="hii";  
String str3="hii";  
System.out.println(str2.equals(str3)); //true
```

>str.equalsIgnoreCase()-->same as equals() method but does not check case

Ex:

```
String str="Hello"  
String str4="hello";  
System.out.println(str.equalsIgnoreCase(str4)); //true
```

>String.valueOf()-->converts any data type of a variable to string

Ex:

```
float a=10.678f;  
String vf=String.valueOf(a);  
System.out.println(vf); //10.678  
System.out.println(vf.concat("wd")); //float is converted to string ,so it prints  
10.678wd
```

>str.toCharArray()-->converts the string to char array

return new char array

Ex:

```
String str5="hellogyuguygk";  
char[] ch=str5.toCharArray();  
for(int i=0;i<ch.length;i++)  
{  
    System.out.println(ch[i]);  
}
```

o/p:

h
e
l
l
o
g
y
u
g
u

y

g

k

>str.substring(i,j) //i and j are indexes
returns part of a string

Here i is mandatory where j can/cannot be given.

Ex:

```
String str5="hellogyuguygk";  
System.out.println("\n"+str5.substring(6)); //yuguygk  
System.out.println("\n"+str5.substring(6,7)); //y
```

>str.split()--> splits this string against given regular expression and returns a char array.

Ex:

```
String s1="Honey is sweet";  
String[] words=s1.split(" ");  
for(String w:words){  
    System.out.println(w);
```

o/p:

Honey

is

sweet

>str.replace(i,j)-->replace all old char to new char in a given string

str.replace(original,replacement)-->replace all original char sequences with replacement characters

Ex:

```
String s1="Honey is sweet";
```

```
System.out.println(s1.replace("s","t")); //Honey it tweet
```

```
System.out.println(s1.replace("is","was")); //Honey was sweet
```

>str.lastIndexOf()-->returns the last index of given character

return -1, if char is not present in the string

Ex:

```
String str="hello"
```

```
System.out.println(str.lastIndexOf("l")); //3
```

```
System.out.println(str.lastIndexOf("z")); //-1
```

>String.join(delimiter,elements)-->returns a joint string with given delimiter.

Ex:

```
String[] str10={"h1","h2"};
String str12=String.join("/",str10);
System.out.println(str12);
System.out.println(String.join("!", "Honey", "is", "sweet")); //Honey!is!sweet
```

o/p:

h1/h2

>str.isEmpty()-->checks whether the string is empty or not.

returns true,if length is 0 else returns false.

Ex:

```
String str6="yug";
String str7="";
System.out.println(str6.isEmpty()); //false
System.out.println(str6.isEmpty()); //true
```

>str.intern()-->returns the string from String Constant Pool.

Ex:

```
----  
public class InternExample{  
    public static void main(String args[]){  
        String s1=new String("hello");  
        String s2="hello";  
        String s3="hii";  
  
        System.out.println(s1==s2);  
        System.out.println(s2==s3);  
  
        s1=s1.intern();      //checks for "hello" string in  
pool,if exists point to that ref and returns true  
        System.out.println(s1==s2);  
    }  
}
```

equals and ==

equals() method compares the content of the string and returns boolean value(true/false)

where as "==" operator compares the references of the objects and returns boolean value(true/false)

concat and +

>concat() method takes concatenates two strings and return new string object only string length is greater than 0, otherwise it returns same object.

adds the string at the end of given string.

>Using "+" opeator ,always creates new string irrespective of length

we add the string in both sides

```
String str="hello"
```

```
String str2="hii"
```

```
str.concat("hii");           //Hellohii
```

```
str2+str;                  //hiiHello
```

SUPER AND THIS KEYWORDS

SUPER AND THIS KEYWORDS

super and this both are reserved keywords, and cant be used as an identifiers

-->this is used to refer current-class's instance as well as static members.

super is used to refer parent class's instance as well as static members.

-->we can user super and this anywhere except in static area.

-->we can use super and this any number of times.

Ex:Using super and this for modifying variables

```
// Program to illustrate super keyword
```

```
// refers super-class instance

class Parent {
    // instance variable
    int a = 10;

    // static variable
    static int b = 20;

}

class Child extends Parent {
    void rr()
    {
        // referring parent class(i.e, class Parent)
        // instance variable(i.e, a)
        System.out.println(super.a);
        System.out.println(this.a);

        // referring parent class(i.e, class Parent)
        // static variable(i.e, b)
        System.out.println(super.b);
        System.out.println(this.a);
    }
}
```

```
public static void main(String[] args)
{
    // Uncomment this and see here you get
    // Compile Time Error since cannot use 'super'
    // in static context.
    // super.a = 700;
    //this.a=160;
    new Base().rr();
}

}
```

Ex:Using super and this for invoking methods.

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

class Dog extends Animal
{
    void bark()
```

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

SUPER() AND THIS() CONSTRUCTOR CALLS

super() and this() are constructor calls to call super and current class constructors

-->we can super() and this() only in constructors, that to as a first statement only.

-->we cannot use both super() and this() at the same time.

i.e, if we use super() we cant use this() and vice-versa

-->Both parent and child should have similar type of constructors, otherwise it throws a mis-match exception.

-->Default(no arg) constructor is called only, when no constructor is present inside the class

-->If any constructors present inside the class, particular constructor will be called based on the arguments specified during th call.

Ex:(USING super() and this() constructors)

```
class Animal
{
    Animal()
    {
        System.out.println("Animal is created");
    }
    Animal(String animal)
    {
        System.out.println("Animal2 is created");
    }
}
class Dog extends Animal
```

```

{
    Dog()
    {
        super("animal");
        //this("cat");
        System.out.println("dog is created");
    }
    Dog(String dog)
    {
        System.out.println("cat is created");
    }
}

class TestSuper3
{
    public static void main(String args[])
    {
        Dog d=new Dog();
    }
}

```

THREADS IN JAVA

Just because the threads share the common **memory** space. The memory allocated to the main thread will be shared by all other child threads.

Whereas in case of Process, the child process are in need to allocate the separate memory **space**.

The main difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces. Lets see the differences in detail:

Thread vs Process

- 1) A program in execution is often referred as process. A thread is a subset(part) of the process.
- 2) A process consists of multiple threads. A thread is a smallest part of the process that can execute concurrently with other parts(threads) of the process.
- 3) A process is sometime referred as task. A thread is often referred as lightweight process.

A process has its own address space. A thread uses the process's address space and share it with the other threads of that process.

A thread can communicate with other thread (of the same process) directly by using methods like `wait()`, `notify()`, `notifyAll()`. A process can communicate with other process by using [inter-process communication](#).

- 7) New threads are easily created. However the creation of new processes require duplication of the parent process.
- 8) Threads have control over the other threads of the same process. A process does not have control over the sibling process, it has control over its child processes only.

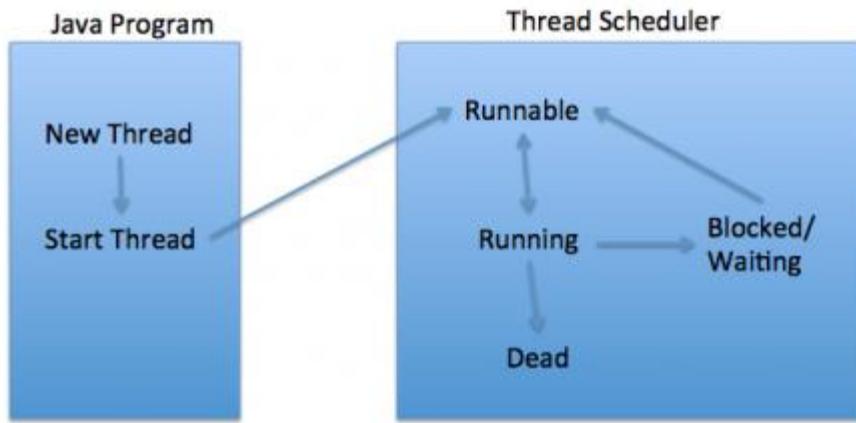
REFERENCE LINK:

<https://beginnersbook.com/2015/01/what-is-the-difference-between-a-process-and-a-thread-in-java/>

Thread Life Cycle in Java

Below diagram shows different states of thread life cycle in java. We can create a thread in java and start it but how the thread states change from

Runnable to Running to Blocked depends on the OS implementation of thread scheduler and java doesn't have full control on that.



New

When we create a new Thread object using `new` operator, thread state is New Thread. At this point, thread is not alive and it's a state internal to Java programming.

Runnable

When we call `start()` function on Thread object, its state is changed to Runnable. The control is given to Thread scheduler to finish its execution. Whether to run this thread instantly or keep it in runnable thread pool before running, depends on the OS implementation of thread scheduler.

Running

When thread is executing, its state is changed to Running. Thread scheduler picks one of the threads from the runnable thread pool and changes its state to Running. Then CPU starts executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of `run()` method or waiting for some resources.

Blocked/Waiting

A thread can be waiting for other thread to finish using [thread join](#) or it can be waiting for some resources to available. For example [producer consumer problem](#) or [waiter notifier implementation](#) or IO resources, then it's state is changed to Waiting. Once the thread wait state is over, it's state is changed to Runnable and it's moved back to runnable thread pool.

Dead

Once the thread finished executing, it's state is changed to Dead and it's considered to be not alive.

- Because it exists normally. This happens when the code of thread has entirely executed by the program.
- Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

A thread that lies in a terminated state does no longer consumes any cycles of CPU.

Thread creation in Java

Thread implementation in java can be achieved in two ways:

1. Extending the `java.lang.Thread` class
2. Implementing the `java.lang.Runnable` Interface

Note: The Thread and Runnable are available in the `java.lang.*` package

1) By extending thread class

- The class should extend Java Thread class.
- The class should override the `run()` method.
- The functionality that is expected by the Thread to be executed is written in the `run()` method.

`void start():` Creates a new thread and makes it runnable.

`void run():` The new thread begins its life inside this method.

Example:

```

public class MyThread extends Thread {
    public void run(){
        System.out.println("thread is running...");
    }
    public static void main(String[] args) {
        MyThread obj = new MyThread();
        obj.start();
    }
}

```

2) By Implementing Runnable interface

- The class should implement the Runnable interface
- The class should implement the run() method in the Runnable interface
- The functionality that is expected by the Thread to be executed is put in the run() method

Example:

```

public class MyThread implements Runnable {
    public void run(){
        System.out.println("thread is running..");
    }
    public static void main(String[] args) {
        Thread t = new Thread(new MyThread());
        t.start();
    }
    //or
    MyThread mt=new MyThread();
    Thread t1=new Thread(mt);
    T1.start();
}

```

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

Extends Thread class vs Implements Runnable Interface?

- Extending the Thread class will make your class unable to extend other classes, because of the single inheritance feature in JAVA. However, this will give you a simpler code structure. If you implement Runnable, you can gain better object-oriented design and consistency and also avoid the single inheritance problems.
- If you just want to achieve basic functionality of a thread you can simply implement Runnable interface and override run() method. But if you want to do something serious with thread object as it has other methods like suspend(), resume(), ..etc which are not available in Runnable interface then you may prefer to extend the Thread class.

THREAD METHODS:

1. **public void run()**: is used to perform action for a thread.
2. **public void start()**: starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds)**: Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join()**: waits for a thread to die.
5. **public void join(long miliseconds)**: waits for a thread to die for the specified milliseconds.
6. **public int getPriority()**: returns the priority of the thread.
7. **public int setPriority(int priority)**: changes the priority of the thread.
8. **public String getName()**: returns the name of the thread.
9. **public void setName(String name)**: changes the name of the thread.
10. **public Thread currentThread()**: returns the reference of currently executing thread.
11. **public int getId()**: returns the id of the thread.
12. **public Thread.State getState()**: returns the state of the thread.
13. **public boolean isAlive()**: tests if the thread is alive.
14. **public void yield()**: causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend()**: is used to suspend the thread(deprecated).
16. **public void resume()**: is used to resume the suspended thread(deprecated).
17. **public void stop()**: is used to stop the thread(deprecated).
18. **public boolean isDaemon()**: tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b)**: marks the thread as daemon or user thread.
20. **public void interrupt()**: interrupts the thread.
21. **public boolean isInterrupted()**: tests if the thread has been interrupted.
22. **public static boolean interrupted()**: tests if the current thread has been interrupted.

REFERENCE LINK:

<https://beginnersbook.com/2013/03/java-threads/>

EXAMPLE FOR SLEEP():

1. **class TestSleepMethod1 extends Thread{**

```
2. public void run(){
3.     for(int i=1;i<5;i++){
4.         try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
5.         System.out.println(i);
6.     }
7. }
8. public static void main(String args[]){
9.     TestSleepMethod1 t1=new TestSleepMethod1();
10.    TestSleepMethod1 t2=new TestSleepMethod1();
11.
12.    t1.start();
13.    t2.start();
14. }
15. }
```

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and starts executing

CAN WE START A THREAD TWICE

No. After starting a thread, it can never be started again. If you do so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

```
1. public class TestThreadTwice1 extends Thread{
2.     public void run(){
3.         System.out.println("running... ");
4.     }
5.     public static void main(String args[]){
6.         TestThreadTwice1 t1=new TestThreadTwice1();
7.         t1.start();
8.         t1.start();
9.     }
10. }
```

WHY DON'T WE CALL RUN() METHOD DIRECTLY, WHY CALL START() METHOD

We can call run() method if we want but then it would behave just like a normal method and we would not be able to take the advantage of [multithreading](#). When the run method gets called through start() method then a new separate thread is being allocated to the execution of run method, so if more than one thread calls start() method that means their run method is being executed by separate threads (these threads run simultaneously).

On the other hand if the run() method of these threads are being called directly then the execution of all of them is being handled by the same current thread and no multithreading will take place, hence the output would reflect the sequential execution of threads in the specified order. Did it confuse you? Lets have a look at the below code to understand this situation.

```
public class RunMethodExample implements Runnable{
    public void run(){
        for(int i=1;i<=3;i++){
            try{
                Thread.sleep(1000);
            }catch(InterruptedException ie){
                ie.printStackTrace();
            }
            System.out.println(i);
        }
    }
    public static void main(String args[]){
        Thread th1 = new Thread(new RunMethodExample(), "th1");
        Thread th2 = new Thread(new RunMethodExample(), "th2");
        th1.run();
        th2.run();
    }
}
```

Output:

```
1
2
3
1
2
```

REFERENCE LINK:

<https://beginnersbook.com/2015/03/why-dont-we-call-run-method-directly-why-call-start-method/>

Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. We can change the name of the thread by using `setName()` method. The syntax of `setName()` and `getName()` methods are given below:

1. **public String getName():** is used to return the name of a thread.
2. **public void setName(String name):** is used to change the name of a thread.

Thread Scheduler in Java

Thread scheduler in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Difference between preemptive scheduling and time slicing

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

Features :

1. The JVM schedules using a preemptive , priority ♦ based scheduling algorithm.
2. All Java threads have a priority and the thread with the highest priority is scheduled to run by the JVM.

3. In case two threads have the same priority a FIFO ordering is followed.

A different thread is invoked to run in case one of the following events occur:

- 1.The currently running thread exits the Runnable state ie either blocks or terminates.
2. A thread with a higher priority than the thread currently running enters the Runnable state. The lower priority thread is preempted and the higher priority thread is scheduled to run.

Time Slicing is dependent on the implementation.

A thread can voluntarily yield control through the `yield()` method. Whenever a thread yields control of the CPU another thread of the same priority is scheduled to run. A thread voluntarily yielding control of the CPU is called Cooperative Multitasking.

Thread Priorities

JVM selects to run a Runnable thread with the highest priority.

All Java threads have a priority in the range 1-10.

Top priority is 10, lowest priority is 1.Normal priority ie. priority by default is 5.

`Thread.MIN_PRIORITY` - minimum thread priority

`Thread.MAX_PRIORITY` - maximum thread priority

`Thread.NORM_PRIORITY` - maximum thread priority

Whenever a new Java thread is created it has the same priority as the thread which created it.

Thread priority can be changed by the `setPriority()` method.

- When a Java thread is created, it inherits its priority from the thread that created it.

- You can modify a thread's priority at any time after its creation using the setPriority method.
- Thread priorities are integers ranging between MIN_PRIORITY (1) and MAX_PRIORITY (10) . The higher the integer, the higher the priority.Normally the thread priority will be 5.

EXAMPLE FOR CHANGING PRIORITY :

```

• class TestMultiPriority1 extends Thread{
•   public void run(){
•     System.out.println("running thread name is:"+Thread.currentThread().get
Name());
•     System.out.println("running thread priority is:"+Thread.currentThread().g
etPriority());
•   }
•   public static void main(String args[]){
•     TestMultiPriority1 m1=new TestMultiPriority1();
•     TestMultiPriority1 m2=new TestMultiPriority1();
•     m1.setPriority(Thread.MIN_PRIORITY);
•     m2.setPriority(Thread.MAX_PRIORITY);
•     m1.start();
•     m2.start();
•   }
• }
```

Output:

Output:running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1

THREAD JOIN() METHOD IN JAVA WITH EXAMPLE

The join() method is used to hold the execution of currently running thread until the specified thread is dead(finished execution). In this tutorial we will discuss the purpose and use of join() method with examples.

Why we use join() method?

In normal circumstances we generally have more than one thread, thread scheduler schedules the threads, which does not guarantee the order of execution of threads.

For example lets have a look at the following code:

Without using join()

Here we have three threads th1, th2 and th3. Even though we have started the threads in a sequential manner the thread scheduler does not start and end them in the specified order. Everytime you run this code, you may get a different result each time. **So the question is: How can we make sure that the threads executes in a particular order. The Answer is: By using join() method appropriately.**

1. **join():** It will put the current thread on wait until the thread on which it is called is dead. If thread is interrupted then it will throw InterruptedException.

Syntax: public final void join()

2. **join(long millis)** :It will put the current thread on wait until the thread on which it is called is dead or wait for specified time (milliseconds).

Syntax: public final synchronized void join(long millis)

3. **join(long millis, int nanos)**: It will put the current thread on wait until the thread on which it is called is dead or wait for specified time (milliseconds + nanos).

Syntax:

```
public final synchronized void join(long millis, int nanos)
```

```
public class JoinExample2 {  
    public static void main(String[] args) {  
        Thread th1 = new Thread(new MyClass2(), "th1");  
        Thread th2 = new Thread(new MyClass2(), "th2");  
        Thread th3 = new Thread(new MyClass2(), "th3");  
  
        th1.start();  
        th2.start();  
        th3.start();  
    }  
}
```

```

}

class MyClass2 implements Runnable{

    @Override
    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("Thread started: "+t.getName());
        try {
            Thread.sleep(4000);
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
        System.out.println("Thread ended: "+t.getName());
    }
}

```

Output:

```

Thread started: th1
Thread started: th3
Thread started: th2
Thread ended: th1
Thread ended: th3
Thread ended: th2

```

Lets have a look at the another code where we are using the join() method.

THE SAME EXAMPLE WITH JOIN()

Lets say our requirement is to execute them in the order of first, second and third. We can do so by using join() method appropriately.

```

public class JoinExample {
    public static void main(String[] args) {
        Thread th1 = new Thread(new MyClass(), "th1");
        Thread th2 = new Thread(new MyClass(), "th2");
        Thread th3 = new Thread(new MyClass(), "th3");

        // Start first thread immediately
        th1.start();

        /* Start second thread(th2) once first thread(th1)
         * is dead
        */
    }
}

```

```

/*
try {
    th1.join();
} catch (InterruptedException ie) {
    ie.printStackTrace();
}
th2.start();

/* Start third thread(th3) once second thread(th2)
 * is dead
*/
try {
    th2.join();
} catch (InterruptedException ie) {
    ie.printStackTrace();
}
th3.start();

// Displaying a message once third thread is dead
try {
    th3.join();
} catch (InterruptedException ie) {
    ie.printStackTrace();
}
System.out.println("All three threads have finished execution");
}
}

```

```

class MyClass implements Runnable{

@Override
public void run() {
    Thread t = Thread.currentThread();
    System.out.println("Thread started: "+t.getName());
    try {
        Thread.sleep(4000);
    } catch (InterruptedException ie) {
        ie.printStackTrace();
    }
    System.out.println("Thread ended: "+t.getName());
}
}

```

Output:

```
Thread started: th1
Thread ended: th1
Thread started: th2
Thread ended: th2
Thread started: th3
Thread ended: th3
All three threads have finished execution
```

REFERENCE LINK:

<https://beginnersbook.com/2015/03/thread-join-method-in-java-with-example/>

THREAD GROUP:

ThreadGroup creates a group of threads. It offers a convenient way to manage groups of threads as a unit. This is particularly valuable in situation in which you want to suspend and resume a number of related threads.

- The thread group form a tree in which every thread group except the initial thread group has a parent.
- A thread is allowed to access information about its own thread group but not to access information about its thread group's parent thread group or any other thread group.

Constructors:

1. **public ThreadGroup(String name):** Constructs a new thread group. The parent of this new group is the thread group of the currently running thread.

Throws: SecurityException - if the current thread cannot create a thread in the specified thread group.

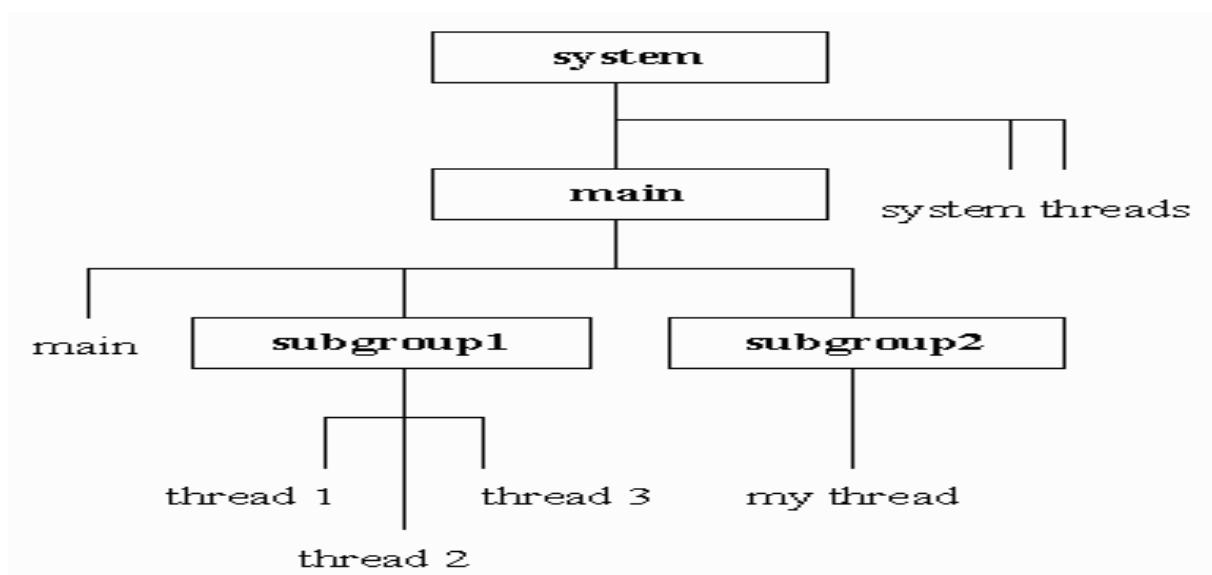
2. **public ThreadGroup(ThreadGroup parent, String name):** Creates a new thread group. The parent of this new group is the specified thread group.
3. **Throws:**
4. NullPointerException - if the thread group argument is null.
5. SecurityException - if the current thread cannot create a thread in the specified thread group.

```
public static void main (String [] args)
```

```
{
```

```
ThreadGroup tg1 = new ThreadGroup ("A");
ThreadGroup tg2 = new ThreadGroup (tg1, "B");
}
```

In the code above, the main thread creates two thread groups: **A** and **B**. First, the main thread creates **A** by calling `ThreadGroup(String name)`. The `tg1`-referenced thread group's parent is `main` because `main` is the main thread's thread group. Second, the main thread creates **B** by calling `ThreadGroup(ThreadGroup parent, String name)`. The `tg2`-referenced thread group's parent is **A** because `tg1`'s reference passes as an argument to `ThreadGroup (tg1, "B")` and **A** associates with `tg1`.



At the top of the figure's structure is the **system** thread group. The JVM-created **system** group organizes JVM threads that deal with object finalization and other system tasks, and serves as the root thread group of an application's hierarchical thread-group structure. Just below **system** is the JVM-created **main** thread group, which is **system**'s subthread group (subgroup, for short). **main** contains at least one thread—the JVM-created main thread that executes byte-code instructions in the **main()** method.

Below the **main** group reside the **subgroup 1** and **subgroup 2** subgroups, application-created subgroups (which the figure's application creates). Furthermore, **subgroup 1** groups three application-created threads: **thread 1**, **thread 2**, and **thread 3**. In contrast, **subgroup 2** groups one application-created thread: **my thread**.

Now that you know the basics, let's start creating thread groups.

Create thread groups and associate threads with those groups
The **ThreadGroup** class's SDK documentation reveals two
constructors: **ThreadGroup(String name)** and **ThreadGroup(ThreadGroup parent, String name)**. Both constructors create a thread group and give it a name, as
the **name** parameter specifies. The constructors differ in their choice of what
thread group serves as parent to the newly created thread group. Each thread
group, except **system**, must have a parent thread group. For **ThreadGroup(String name)**, the parent is the thread group of the thread that calls **ThreadGroup(String name)**. As an example, if the main thread calls **ThreadGroup(String name)**, the
newly created thread group has the main thread's group as its parent—**main**.
For **ThreadGroup(ThreadGroup parent, String name)**, the parent is the group
that **parent** references. The following code shows how to use these constructors
to create a pair of thread groups:

[[Learn Java from beginning concepts to advanced design patterns in this comprehensive 12-part course!](#)]

```
public static void main (String [] args)
{
    ThreadGroup tg1 = new ThreadGroup ("A");
    ThreadGroup tg2 = new ThreadGroup (tg1, "B");
}
```

In the code above, the main thread creates two thread groups: **A** and **B**. First, the
main thread creates **A** by calling **ThreadGroup(String name)**. The **tg1**-referenced
thread group's parent is **main** because **main** is the main thread's thread group.
Second, the main thread creates **B** by calling **ThreadGroup(ThreadGroup parent, String name)**. The **tg2**-referenced thread group's parent is **A** because **tg1**'s
reference passes as an argument to **ThreadGroup (tg1, "B")** and **A** associates
with **tg1**.

Tip: Once you no longer need a hierarchy of **ThreadGroup** objects,
call **ThreadGroup's void destroy()** method via a reference to
the **ThreadGroup** object at the top of that hierarchy. If the
top **ThreadGroup** object and all subgroup objects lack thread
objects, **destroy()** prepares those thread group objects for garbage collection.
Otherwise, **destroy()** throws an **IllegalThreadStateException** object. However,
until you nullify the reference to the top **ThreadGroup** object (assuming a field
variable contains that reference), the garbage collector cannot collect that object.
Referencing the top object, you can determine if a previous call was made to

the `destroy()` method by calling `ThreadGroup`'s `boolean isDestroyed()` method. That method returns true if the thread group hierarchy was destroyed.

By themselves, thread groups are useless. To be of any use, they must group threads. You group threads into thread groups by passing `ThreadGroup` references to appropriate `Thread` constructors:

```
ThreadGroup tg = new ThreadGroup ("subgroup 2");
Thread t = new Thread (tg, "my thread");
```

The code above first creates a `subgroup 2` group with `main` as the parent group. (I assume the main thread executes the code.) The code next creates a `my thread` `Thread` object in the `subgroup 2` group.

Now, let's create an application that produces our figure's hierarchical thread-group structure:

Listing 1. ThreadGroupDemo.java

```
// ThreadGroupDemo.java
class ThreadGroupDemo
{
    public static void main (String [] args)
    {
        ThreadGroup tg = new ThreadGroup ("subgroup 1");
        Thread t1 = new Thread (tg, "thread 1");
        Thread t2 = new Thread (tg, "thread 2");
        Thread t3 = new Thread (tg, "thread 3");
        tg = new ThreadGroup ("subgroup 2");
        Thread t4 = new Thread (tg, "my thread");
        tg = Thread.currentThread ().getThreadGroup ();
        int agc = tg.activeGroupCount ();
        System.out.println ("Active thread groups in " + tg.getName () +
                           " thread group: " + agc);
        tg.list ();
    }
}
```

`ThreadGroupDemo` creates the appropriate thread group and thread objects to mirror what you see in the figure above. To prove that the `subgroup 1` and `subgroup 2` groups are `main`'s only subgroups, `ThreadGroupDemo` does the following:

1. Retrieves a reference to the main thread's `ThreadGroup` object by calling `Thread`'s static `currentThread()` method (which returns a reference to the main thread's `Thread` object) followed by `Thread`'s `ThreadGroup getThreadGroup()` method.
2. Calls `ThreadGroup`'s `int activeGroupCount()` method on the just-returned `ThreadGroup` reference to return an estimate of active groups within the main thread's thread group.
3. Calls `ThreadGroup`'s `String getName ()` method to return the main thread's thread group name.
4. Calls `ThreadGroup`'s `void list ()` method to print on the standard output device details on the main thread's thread group and all subgroups.
5. **int activeCount():** This method returns the number of threads in the group plus any group for which this thread is parent.

Syntax: public int activeCount()

Returns: This method returns an estimate of the number of active threads in this thread group and in any other thread group that has this thread group as an ancestor.

Exception: NA

6. **int activeGroupCount():** This method returns an estimate of the number of active groups in this thread group.

Syntax: public int activeGroupCount().

Returns: Returns the number of groups for which the invoking thread is parent.

Exception: NA.

- `destroy()`: destroys the thread group and all of its subgroups.
- `enumerate(Thread[] list)`: copies into the specified array every active thread in this thread group and its subgroups.
- `getMaxPriority()`: returns the maximum priority of the thread group.
- `interrupt()`: interrupts all threads in the thread group.

- `isDaemon()`: tests if the thread group is a daemon thread group.
- `setMaxPriority(int priority)`: sets the maximum priority of the group.

PRIORITY AND THREAD GROUPS

A thread group's maximum priority is the highest priority any of its threads can attain. Consider the aforementioned network server program. Within that program, a thread waits for and accepts requests from client programs. Before doing that, the wait-for/accept-request thread might first create a thread group with a maximum priority just below that thread's priority. Later, when a request arrives, the wait-for/accept-request thread creates a new thread to respond to the client request and adds the new thread to the previously created thread group. The new thread's priority automatically lowers to the thread group's maximum. That way, the wait-for/accept-request thread responds more often to requests because it runs more often.

Java assigns a maximum priority to each thread group. When you create a group, Java obtains that priority from its parent group. Use `ThreadGroup's void setMaxPriority(int priority)` method to subsequently set the maximum priority. Any threads that you add to the group after setting its maximum priority cannot have a priority that exceeds the maximum. Any thread with a higher priority automatically lowers when it joins the thread group. However, if you use `setMaxPriority(int priority)` to lower a group's maximum priority, all threads added to the group prior to that method call keep their original priorities. For example, if you add a priority 8 thread to a maximum priority 9 group, and then lower that group's maximum priority to 7, the priority 8 thread remains at priority 8. At any time, you can determine a thread group's maximum priority by calling `ThreadGroup's int getMaxPriority()` method. To demonstrate priority and thread groups, I wrote `MaxPriorityDemo`:

Listing 2. MaxPriorityDemo.java

```
// MaxPriorityDemo.java
class MaxPriorityDemo
{
    public static void main (String [] args)
    {
        ThreadGroup tg = new ThreadGroup ("A");
        System.out.println ("tg maximum priority = " + tg.getMaxPriority());
```

```

Thread t1 = new Thread (tg, "X");
System.out.println ("t1 priority = " + t1.getPriority ());
t1.setPriority (Thread.NORM_PRIORITY + 1);
System.out.println ("t1 priority after setPriority() = " +
    t1.getPriority ());
tg.setMaxPriority (Thread.NORM_PRIORITY - 1);
System.out.println ("tg maximum priority after setMaxPriority() = " +
    tg.getMaxPriority ());
System.out.println ("t1 priority after setMaxPriority() = " +
    t1.getPriority ());
Thread t2 = new Thread (tg, "Y");
System.out.println ("t2 priority = " + t2.getPriority ());
t2.setPriority (Thread.NORM_PRIORITY);
System.out.println ("t2 priority after setPriority() = " +
    t2.getPriority ());
}
}

```

When run, **MaxPriorityDemo** produces the following output:

```

tg maximum priority = 10
t1 priority = 5
t1 priority after setPriority() = 6
tg maximum priority after setMaxPriority() = 4
t1 priority after setMaxPriority() = 6
t2 priority = 4
t2 priority after setPriority() = 4

```

Thread group A (which tg references) starts with the highest priority (10) as its maximum. Thread X, whose Thread object t1 references, joins the group and receives 5 as its priority. We change that thread's priority to 6, which succeeds because 6 is less than 10. Subsequently, we call **setMaxPriority(int priority)** to reduce the group's maximum priority to 4. Although thread X remains at priority 6, a newly-added Y thread receives 4 as its priority. Finally, an attempt to increase thread Y's priority to 5 fails, because 5 is greater than 4.

Note: **setMaxPriority(int priority)** automatically adjusts the maximum priority of a thread group's subgroups.

In addition to using thread groups to limit thread priority, you can accomplish other tasks by calling various `ThreadGroup` methods that apply to each group's thread. Methods include `void suspend()`, `void resume()`, `void stop()`, and `void interrupt()`. Because Sun Microsystems has deprecated the first three methods (they are unsafe), we examine only `interrupt()`.

REFERENCE LINK:

<https://www.journaldev.com/1016/java-thread-example>

Daemon thread in Java

Daemon thread is a low priority thread that runs in background to perform tasks such as garbage collection.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

Properties:

- They can not prevent the JVM from exiting when all the user threads finish their execution.
- JVM terminates itself when all user threads finish their execution
- If JVM finds running daemon thread, it terminates the thread and after that shutdown itself. JVM does not care whether Daemon thread is running or not.
- It is an utmost low priority thread.

Methods:

1. **void setDaemon(boolean status)**: This method is used to mark the current thread as daemon thread or user thread. For example if I have a user thread tU then tU.setDaemon(true) would make it Daemon thread. On the other hand if I have a Daemon thread tD then by calling tD.setDaemon(false) would make it user thread.

Syntax:

2. **public final void setDaemon(boolean on)**
3. **parameters:**
4. **on** : if true, marks this thread as a daemon thread.
5. **exceptions:**
6. **IllegalThreadStateException**: if only this thread is active.
SecurityException: if the current thread cannot modify this thread.
7. **boolean isDaemon()**:

This method is used to check that current is daemon. It returns true if the

thread is Daemon else it returns false.

Syntax:

8. **public final boolean isDaemon()**
9. **returns:**
10. This method returns true if this thread is a daemon thread; false otherwise

If you call the setDaemon() method after starting the thread, it would throw **IllegalThreadStateException**.

filter_none
edit
play_arrow
brightness_4

```
// Java program to demonstrate the usage of
// exception in Daemon() Thread
public class DaemonThread extends Thread
{
    public void run()
    {
        System.out.println("Thread name: " + Thread.currentThread().getName());
        System.out.println("Check if its DaemonThread: "
                           + Thread.currentThread().isDaemon());
    }

    public static void main(String[] args)
    {
        DaemonThread t1 = new DaemonThread();
        DaemonThread t2 = new DaemonThread();
        t1.start();

        // Exception as the thread is already started
        t1.setDaemon(true);

        t2.start();
    }
}
```

Runtime exception:

```
Exception in thread "main" java.lang.IllegalThreadStateException
```

```
at java.lang.Thread.setDaemon(Thread.java:1352)
```

```
at DaemonThread.main(DaemonThread.java:19)
```

Output:

```
Thread name: Thread-0
```

```
Check if its DaemonThread: false
```

This clearly shows that we cannot call the setDaemon() method after starting the thread.

Daemon vs User Threads

1. **Priority:** When the only remaining threads in a process are daemon threads, the interpreter exits. This makes sense because when only daemon threads remain, there is no other thread for which a daemon thread can provide a service.
2. **Usage:** Daemon thread is to provide services to user thread for background supporting task

Java Thread enumerate() method

The **enumerate()** method of thread class is used to copy every active thread's thread group and its subgroup into the specified array. This method calls the enumerate method with the **tarray** argument.

This method uses the activeCount method to get an estimate of how big the array should be. If the length of the array is too short to hold all the threads, the extra threads are silently ignored.

Syntax

1. **public static int enumerate(Thread[] tarray)**

Parameter

tarray: This method is an array of Thread objects to copy to.

Return

This method returns the number of threads put into the array.

yield()

We can prevent the execution of a thread by using one of the following methods of Thread class.

yield(): Suppose there are three threads t1, t2, and t3. Thread t1 gets the processor and starts its execution and thread t2 and t3 are in Ready/Runnable

state. Completion time for thread t1 is 5 hour and completion time for t2 is 5 minutes. Since t1 will complete its execution after 5 hours, t2 has to wait for 5 hours to just finish 5 minutes job. In such scenarios where one thread is taking too much time to complete its execution, we need a way to prevent execution of a thread in between if something important is pending. `yield()` helps us in doing so.

yield() basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.

Java Thread `isAlive()` method

The **isAlive()** method of thread class tests if the thread is alive. A thread is considered alive when the `start()` method of thread class has been called and the thread is not yet dead. This method returns true if the thread is still running and not finished.

Syntax

1. `public final boolean isAlive()`

Return

This method will return true if the thread is alive otherwise returns false.

Example

```
1. public class JavalsAliveExp extends Thread
2. {
3.     public void run()
4.     {
5.         try
6.         {
7.             Thread.sleep(300);
8.             System.out.println("is run() method isAlive "+Thread.currentThread().isAlive());
9.         }
10.        catch (InterruptedException ie) {
11.            }
12.        }
13.        public static void main(String[] args)
14.        {
15.            JavalsAliveExp t1 = new JavalsAliveExp();
16.            System.out.println("before starting thread isAlive: "+t1.isAlive());
17.            t1.start();
```

```
18.         System.out.println("after starting thread isAlive: "+t1.isAlive());  
19.     }  
20. }
```

Output:

```
before starting thread isAlive: false  
after starting thread isAlive: true  
is run() method isAlive true
```

Java Thread suspend() method

The **suspend()** method of thread class puts the thread from running to waiting state. This method is used if you want to stop the thread execution and start it again when a certain event occurs. This method allows a thread to temporarily cease execution. The suspended thread can be resumed using the resume() method.

Syntax

1. **public final void** suspend()

Return

This method does not return any value.

Exception

SecurityException: If the current thread cannot modify the thread.

```
1. public class JavaSuspendExp extends Thread  
2. {  
3.     public void run()  
4.     {  
5.         for(int i=1; i<5; i++)  
6.         {  
7.             try  
8.             {  
9.                 // thread to sleep for 500 milliseconds  
10.                sleep(500);  
11.                System.out.println(Thread.currentThread().getName());  
12.            }catch(InterruptedException e){System.out.println(e);}  
13.        }  
14.    }  
15. }
```

```

13.         System.out.println(i);
14.     }
15. }
16. public static void main(String args[])
17. {
18.     // creating three threads
19.     JavaSuspendExp t1=new JavaSuspendExp ();
20.     JavaSuspendExp t2=new JavaSuspendExp ();
21.     JavaSuspendExp t3=new JavaSuspendExp ();
22.     // call run() method
23.     t1.start();
24.     t2.start();
25.     // suspend t2 thread
26.     t2.suspend();
27.     // call run() method
28.     t3.start();
29. }
30. }
```

[Test it Now](#)

Output:

```

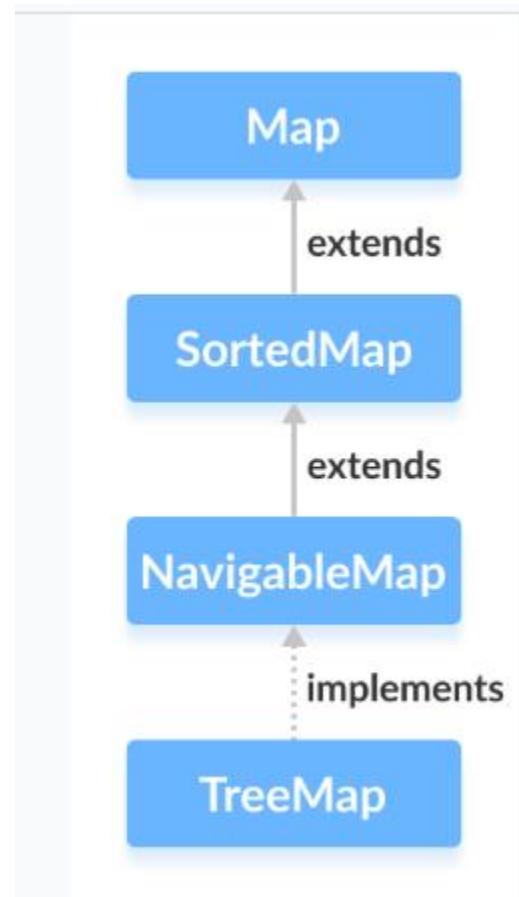
Thread-0
1
Thread-2
1
Thread-0
2
Thread-2
2
Thread-0
3
Thread-2
3
Thread-0
4
Thread-2
4
```

TREEMAP IN JAVA

- The TreeMap in Java is used to implement Map interface and NavigableMap along with the Abstract Class.

- The TreeMap class of the Java collections framework provides the tree data structure implementation.
- map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.
- This proves to be an efficient way of sorting and storing the key-value pairs.
- It is a red-Black tree based NavigableMap implementation.
- TreeMap is not synchronized and thus is not thread-safe. For multithreaded environments, accidental unsynchronized access to the map is prevented by:

```
SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));
```



TreeMap Declaration:

In order to create a TreeMap, we must import the `java.util.TreeMap` package first.

```
public class TreeMap<K,V> extends AbstractMap<K,V> implements NavigableMap<K,V>, Cloneable, Serializable
```

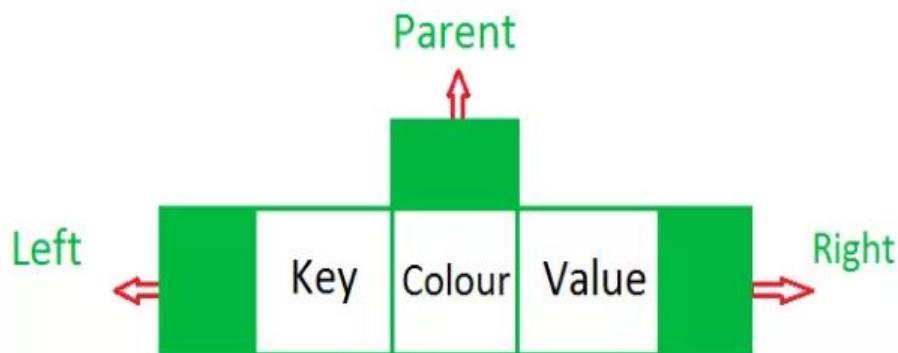
Here,

K: It is the type of keys maintained by this map.

V: It is the type of mapped values.

Internal Structure of TreeMap:

TreeMap is based on tree data structure as its name suggested. As we know that, in a tree, each node has three references its parent, right and left element. Let's see the following diagram:



TreeMap is based upon tree data structure. Each node in the tree has,

- 3 Variables (*K key=Key, V value=Value, boolean color=Color*)
- 3 References (*Entry left = Left, Entry right = Right, Entry parent = Parent*)

<https://www.dineshonjava.com/internal-working-of-treemap-in-java/>

<https://www.youtube.com/watch?v=qA02XWRTBdw>

Features of TreeMap:

- It stores key-value pairs similar to like HashMap.
- It allows only distinct keys. Duplicate keys are not possible.
- It cannot have null key but can have multiple null values.
- It stores the keys in sorted order (natural order) or by a Comparator provided at map creation time.
- It provides guaranteed $\log(n)$ time cost for the containsKey, get, put and remove operations.
- TreeMap has better performance in memory management as it does not maintain an array internally to store key-value pairs.

- TreeMap is implemented using Red-Black tree which consumes more memory compare to hashmap,linkedhashmap.
- The iterators returned by the iterator method are fail-fast.
- TreeMap is iterated according to the natural ordering of its keys, or according to the comparator specified at map's creation time.

Constructors of TreeMap class:

Constructor	Description
TreeMap()	It is used to construct an empty tree map that will be sorted using the natural order of its key.
TreeMap(Comparator<? super K> comparator)	It is used to construct an empty tree-based map that will be sorted using the comparator comp.
TreeMap(Map<? extends K,? extends V> m)	It is used to initialize a treemap with the entries from m , which will be sorted using the natural order of the keys.
TreeMap(SortedMap<K,? extends V> m)	It is used to initialize a treemap with the entries from the m in the same order as sm .

<https://www.geeksforgeeks.org/treemap-in-java/>

Methods of TreeMap:

Map.Entry<K,V> ceilingEntry(K key)	It returns the key-value pair having the least key, greater than or equal to the specified key, or null if there is no such key.
K ceilingKey(K key)	It returns the least key, greater than the specified key or null if there is no such key.
void clear()	It removes all the key-value pairs from a map.
Object clone()	It returns a shallow copy of TreeMap instance.
Comparator<? super K> comparator()	It returns the comparator that arranges the key in order, or null if the map uses the natural ordering.

NavigableSet<K> descendingKeySet()	It returns a reverse order NavigableSet view of the keys contained in the map.
NavigableMap<K,V> descendingMap()	It returns the specified key-value pairs in descending order.
Map.Entry firstEntry()	It returns the key-value pair having the least key.
Map.Entry<K,V> floorEntry(K key)	It returns the greatest key, less than or equal to the specified key, or null if there is no such key.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
SortedMap<K,V> headMap(K toKey)	It returns the key-value pairs whose keys are strictly less than toKey.
NavigableMap<K,V> headMap(K toKey, boolean inclusive)	It returns the key-value pairs whose keys are less than (or equal to if inclusive is true) toKey.
Map.Entry<K,V> higherEntry(K key)	It returns the least key strictly greater than the given key, or null if there is no such key.
K higherKey(K key)	It is used to return true if this map contains a mapping for the specified key.
Set keySet()	It returns the collection of keys exist in the map.
Map.Entry<K,V> lastEntry()	It returns the key-value pair having the greatest key, or null if there is no such key.
Map.Entry<K,V> lowerEntry(K key)	It returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key.
K lowerKey(K key)	It returns the greatest key strictly less than the given key, or null if there is no such key.

NavigableSet<K> navigableKeySet()	It returns a NavigableSet view of the keys contained in this map.
Map.Entry<K,V> pollFirstEntry()	It removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
Map.Entry<K,V> pollLastEntry()	It removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
V put(K key, V value)	It inserts the specified value with the specified key in the map.
void putAll(Map<? extends K,? extends V> map)	It is used to copy all the key-value pair from one map to another map.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value only if the old value is already associated with the specified key.
void replaceAll(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)	It returns key-value pairs whose keys range from fromKey to toKey.
SortedMap<K,V> subMap(K fromKey, K toKey)	It returns key-value pairs whose keys range from fromKey, inclusive, to toKey, exclusive.
SortedMap<K,V> tailMap(K fromKey)	It returns key-value pairs whose keys are greater than or equal to fromKey.
NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)	It returns key-value pairs whose keys are greater than (or equal to, if inclusive is true) fromKey.

boolean containsKey(Object key)	It returns true if the map contains a mapping for the specified key.
boolean containsValue(Object value)	It returns true if the map maps one or more keys to the specified value.
K firstKey()	It is used to return the first (lowest) key currently in this sorted map.
V get(Object key)	It is used to return the value to which the map maps the specified key.
K lastKey()	It is used to return the last (highest) key currently in the sorted map.
V remove(Object key)	It removes the key-value pair of the specified key from the map.
Set<Map.Entry<K,V>> entrySet()	It returns a set view of the mappings contained in the map.
int size()	It returns the number of key-value pairs exists in the hashtable.
Collection values()	It returns a collection view of the values contained in the map.

Example:

```
// Java code to show example of TreeMap
```

```
import java.util.*;
import java.util.concurrent.*;
public class Main {
    public static void main(String args[]){
        TreeMap<Integer, String> tree_map = new TreeMap<Integer, String>();
        // Mapping string values to int keys
        tree_map.put(10, "Geeks");
```

```
tree_map.put(15, "4");
tree_map.put(20, "Geeks");
tree_map.put(25, "Welcomes");
tree_map.put(30, null);
tree_map.put(5,"U");

TreeMap tree_map2 = new TreeMap();
tree_map2.put(10,"hello");
tree_map2.put(10,"hii");
tree_map2.put(3,"heyy");
System.out.println(tree_map2);
tree_map2.putAll(tree_map);

System.out.println("TreeMap: "+tree_map2);
// Displaying the TreeMap1
System.out.println("TreeMap: "+tree_map);

System.out.println(tree_map.ceilingEntry(14));
System.out.println(tree_map.floorEntry(14));
System.out.println("CEILING KEY: "+tree_map.ceilingKey(14));
System.out.println("NAVIGABLE KEY SET :" +tree_map.navigableKeySet());
System.out.println("DESCENDING KEY SET: "+tree_map.descendingKeySet());
System.out.println("DESCENDING MAP :" +tree_map.descendingMap());
System.out.println("TreeMap: "+tree_map);

System.out.println(tree_map.firstEntry());
System.out.println(tree_map.lastEntry());
System.out.println(tree_map.higherEntry(20));
System.out.println(tree_map.lowerEntry(20));
System.out.println(tree_map.higherKey(20));
System.out.println(tree_map.lowerKey(20));
```

```

System.out.println("HEAD MAP: "+tree_map.headMap(20));
System.out.println("TAIL MAP: "+tree_map.tailMap(20));
System.out.println("HEAD MAP: "+tree_map.headMap(20,true));
System.out.println("TAIL MAP: "+tree_map.tailMap(20,false));

tree_map.replace(15,"for");
tree_map.replace(15,"for","4");
tree_map.replaceAll((Key, Value) -> Value + 2);

System.out.println("KEY SET :" +tree_map.keySet());
System.out.println("VALUES :" +tree_map.values());
System.out.println(tree_map.containsKey(15));
System.out.println(tree_map.containsValue("Welcomes"));
System.out.println(tree_map2.get(3));
System.out.println(tree_map2.remove(3));
System.out.println(tree_map.size());

// Displaying the TreeMap
System.out.println("TreeMap: " +tree_map.entrySet());
//display using entrySet
for(Map.Entry m:tree_map.entrySet())
{
    System.out.println(m.getKey()+" "+m.getValue());
}
}

}

```

Output:

```

{3=heyy, 10=hii}
TreeMap: {3=heyy, 5=U, 10=Geeks, 15=4, 20=Geeks, 25=Welcomes, 30=null}

```

```
TreeMap: {5=U, 10=Geeks, 15=4, 20=Geeks, 25=Welcomes, 30=null}
15=4
10=Geeks
CEILING KEY: 15
NAVIGABLE KEY SET :[5, 10, 15, 20, 25, 30]
DESCENDING KEY SET: [30, 25, 20, 15, 10, 5]
DESCENDING MAP :{30=null, 25=Welcomes, 20=Geeks, 15=4, 10=Geeks, 5=U}
TreeMap: {5=U, 10=Geeks, 15=4, 20=Geeks, 25=Welcomes, 30=null}
5=U
30=null
25=Welcomes
15=4
25
15
HEAD MAP: {5=U, 10=Geeks, 15=4}
TAIL MAP: {20=Geeks, 25=Welcomes, 30=null}
HEAD MAP: {5=U, 10=Geeks, 15=4, 20=Geeks}
TAIL MAP: {25=Welcomes, 30=null}
KEY SET :[5, 10, 15, 20, 25, 30]
VALUES :[U2, Geeks2, 42, Geeks2, Welcomes2, null2]
true
false
heyy
heyy
6
TreeMap: [5=U2, 10=Geeks2, 15=42, 20=Geeks2, 25=Welcomes2, 30=null2]
5 U2
10 Geeks2
15 42
20 Geeks2
```

25 Welcomes2

30 null2

Treemap Performance:

TreeMap provides the performance of $\log(n)$ for most operations like add(), remove() and contains(). HashMap performs with constant-time performance $O(1)$ for same operations. In that way, HashMap performs much better than TreeMap.

TreeMap has better performance in memory management as it does not maintain an array internally to store key-value pairs. In HashMap, array size is determined while initialization or resizing which if is often more than needed at the time. It waste the memory. There is no such problem with TreeMap.

Difference between hashmap-treemap-linkedhashmap

Property	HashMap	TreeMap	LinkedHashMap	HashTable
Iteration Order	Random	Sorted according to natural order of keys	Sorted according to the insertion order .	Random
Efficiency: Get, Put, Remove, ContainsKey	$O(1)$	$O(\log(n))$	$O(1)$	$O(1)$
Null keys/values	allowed	Not-allowed*	allowed	Not-allowed
Interfaces	Map	Map, SortedMap, NavigableMap	Map	Map
Synchronized	Not instead use <code>Collection.synchronizedMap(new HashMap())</code>			Yes but prefer to use <code>ConcurrentHashMap</code>
Implementation	Buckets	Red-Black tree	HashTable and LinkedList using doubly linked list of buckets	Buckets
Comments	Efficient	Extra cost of maintaining TreeMap	Advantage of TreeMap without extra cost.	Obsolete

CONCLUSION:

- 1.The underlying data structure is RED-BLACK Tree.

- 2.Duplicate keys are not allowed but values can be duplicated.
- 3.Insertion order is not preserved and all entries will be inserted according to some sorting order of keys.
- 4.If we are depending on default natural sorting order keys should be homogeneous and Comparable otherwise we will get ClassCastException.
- 5.If we are defining our own sorting order by Comparator then keys can be heterogeneous and non Comparable.
- 6.There are no restrictions on values they can be heterogeneous and non Comparable.
- 7.For the non empty TreeMap if we are trying to insert an entry with null key we will get NullPointerException.
- 8.There are no restrictions for null values.

REFERENCE LINKS:

<https://www.geeksforgeeks.org/treemap-in-java/>

<https://beginnersbook.com/2013/12/treemap-in-java-with-example/>

<https://www.javatpoint.com/java-treemap>

<https://www.callicoder.com/java-treemap/>

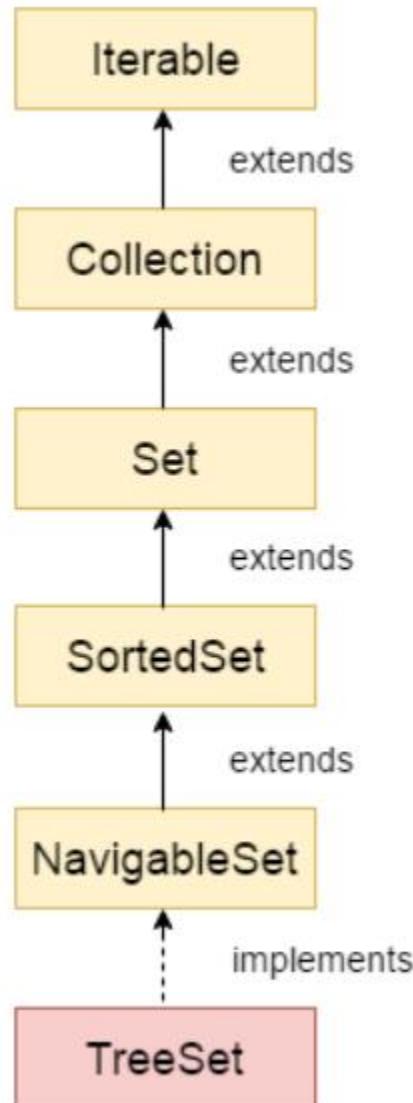
<https://howtodoinjava.com/java/collections/treemap-class/>

<https://www.techiedelight.com/difference-between-hashmap-treemap-linkedhashmap-java/>

TREESET CLASS IN JAVA

- Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

- The sort order is either natural order or by a [Comparator](#) provided at treeset creation time, depending on which [constructor](#) is used.
- Treeset Hierarchy



Declaration:

```

class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, Serializable
{
    //implementation
}
  
```

Treeset Features:

- TreeSet implements the SortedSet interface so duplicate values are not allowed.
- Objects in a TreeSet are stored in a sorted and ascending order.
- TreeSet does not preserve the insertion order of elements but elements are sorted by keys.
- TreeSet does not allow to insert Heterogeneous objects. It will throw classCastException at Runtime if trying to add heterogeneous objects.
- TreeSet serves as an excellent choice for storing large amounts of sorted information which are supposed to be accessed quickly because of its faster access and retrieval time.
- TreeSet is basically implementation of a self-balancing binary search tree like Red-Black Tree.
- TreeSet is not synchronized.
- Use Collections.synchronizedSortedSet(new TreeSet()) method to get the synchronized TreeSet.
- TreeSet provides guaranteed $\log(n)$ time cost for the basic operations (add, remove and contains).
- The operations like iterating the elements in sorted order takes $O(n)$ time.
- TreeSet adds elements to it according to their natural order. This internally compares the elements with each other using the compareTo (or compare) method.
- If you try to compare any object with a null value using one of these methods, a NullPointerException will be thrown.
- TreeSet is very useful collection class in cases where we want to handle duplicate records in sorted manner.

Constructors of Java TreeSet Class:

Constructor	Description
TreeSet()	It is used to construct an empty tree set that will be sorted in ascending order according to the natural order of the tree set.

TreeSet(Collection<? extends E> c)	It is used to build a new tree set that contains the elements of the collection c.
TreeSet(Comparator<? super E> comparator)	It is used to construct an empty tree set that will be sorted according to given comparator.
TreeSet(SortedSet<E> s)	It is used to build a TreeSet that contains the elements of the given SortedSet.

Example:

```

import java.util.Comparator;
import java.util.SortedSet;
import java.util.TreeSet;

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

        //Creating an empty TreeSet

        SortedSet<String> fruits1 = new TreeSet<>();
        fruits1.add("Banana");
        fruits1.add("Apple");
        fruits1.add("Pineapple");
        fruits1.add("Orange");
        System.out.println("Fruits Set : " + fruits1);

        // Duplicate elements are ignored
        fruits1.add("Apple");
    }
}

```

```

        System.out.println("After adding duplicate element \"Apple\" : " +
fruits1);

// This will be allowed because it's in lowercase.
fruits1.add("banana");
System.out.println("After adding \"banana\" : " + fruits1);

// Creating a TreeSet with a custom Comparator (Descending Order)
SortedSet<String> fruits2 = new
TreeSet<>(Comparator.reverseOrder());

// Adding new elements to a TreeSet
fruits2.add("Banana");
fruits2.add("Apple");
fruits2.add("Pineapple");
fruits2.add("Orange");

System.out.println("Fruits Set using Comparator : " + fruits2);

}

}

```

Output:

```

Fruits Set : [Apple, Banana, Orange, Pineapple]
After adding duplicate element "Apple" : [Apple, Banana, Orange, Pineapple]
After adding "banana" : [Apple, Banana, Orange, Pineapple, banana]
Fruits Set using Comparator : [Pineapple, Orange, Banana, Apple]

```

Methods of TreeSet:

Method	Description
boolean add(E e)	It is used to add the specified element to this set if it is not already present.
boolean addAll(Collection<? extends E> c)	It is used to add all of the elements in the specified collection to this set.
E ceiling(E e)	It returns the equal or closest greatest element of the specified element from the set, or null there is no such element.
Comparator<? super E> comparator()	It returns comparator that arranged elements in order, or null if this set uses the natural ordering of its elements.
Iterator descendingIterator()	It is used iterate the elements in descending order.
NavigableSet descendingSet()	It returns the elements in reverse order.
E floor(E e)	It returns the equal or closest least element of the specified element from the set, or null there is no such element.
SortedSet headSet(E toElement)	It returns the group of elements that are less than the specified element.
NavigableSet headSet(E toElement, boolean inclusive)	It returns the group of elements that are less than or equal to(if, inclusive is true) the specified element.
E higher(E e)	It returns the closest greatest element of the specified element from the set, or null there is no such element.
Iterator iterator()	It is used to iterate the elements in ascending order.
E lower(E e)	It returns the closest least element of the specified element from the set, or null there is no such element.

E pollFirst()	It is used to retrieve and remove the lowest(first) element.
E pollLast()	It is used to retrieve and remove the highest(last) element.
Spliterator spliterator()	It is used to create a late-binding and fail-fast spliterator over the elements.
NavigableSet subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	It returns a set of elements that lie between the given range.
SortedSet subSet(E fromElement, E toElement))	It returns a set of elements that lie between the given range which includes fromElement and excludes toElement.
SortedSet tailSet(E fromElement)	It returns a set of elements that are greater than or equal to the specified element.
NavigableSet tailSet(E fromElement, boolean inclusive)	It returns a set of elements that are greater than or equal to (if, inclusive is true) the specified element.
boolean contains(Object o)	It returns true if this set contains the specified element.
boolean isEmpty()	It returns true if this set contains no elements.
boolean remove(Object o)	It is used to remove the specified element from this set if it is present.
void clear()	It is used to remove all of the elements from this set.
Object clone()	It returns a shallow copy of this TreeSet instance.
E first()	It returns the first (lowest) element currently in this sorted set.
E last()	It returns the last (highest) element currently in this sorted set.
int size()	It returns the number of elements in this set.

Difference between pollFirst(),pollLast(),first() and last() methods

```
import java.util.*;
class Main{
    public static void main(String args[]){
        TreeSet<Integer> set=new TreeSet<Integer>();
        set.add(24);
        set.add(66);
        set.add(12);
        set.add(15);
        set.add(3);
        set.add(5);
        set.add(7);
        set.add(8);
        System.out.println(set);

        System.out.println("Lowest Value: "+set.pollFirst());
        System.out.println("Highest Value: "+set.pollLast());
        System.out.println(set);

        System.out.println("Last Value: "+set.last());
        System.out.println(" First Value: "+set.first());
        System.out.println(set);
    }
}
```

Output:

[3, 5, 7, 8, 12, 15, 24, 66]

Lowest Value: 3

```
Highest Value: 66  
[5, 7, 8, 12, 15, 24]  
Last Value: 24  
First Value: 5  
[5, 7, 8, 12, 15, 24]
```

Difference between NavigableSet and SortedSet operations.

```
import java.util.*;  
  
class Main{  
  
    public static void main(String args[]){  
  
        TreeSet<String> set=new TreeSet<String>();  
  
        set.add("A");  
        set.add("B");  
        set.add("C");  
        set.add("D");  
        set.add("E");  
  
        //NavigableSet operations.  
  
        System.out.println("Initial Set: "+set);  
  
        System.out.println("Reverse Set: "+set.descendingSet());  
  
        System.out.println("Head Set: "+set.headSet("C", true));  
  
        System.out.println("SubSet: "+set.subSet("A", false, "D", true));
```

```
System.out.println("TailSet: "+set.tailSet("C", false));  
  
//Sorted set operations  
  
System.out.println("Intial Set: "+set);  
  
System.out.println("Head Set: "+set.headSet("C"));  
  
System.out.println("SubSet: "+set.subSet("A", "D"));  
  
System.out.println("TailSet: "+set.tailSet("C"));  
  
}  
}
```

Output:

```
Initial Set: [A, B, C, D, E]
```

```
Reverse Set: [E, D, C, B, A]
```

```
Head Set: [A, B, C]
```

```
SubSet: [B, C, D]
```

```
TailSet: [D, E]
```

```
Intial Set: [A, B, C, D, E]
```

```
Head Set: [A, B]
```

```
SubSet: [A, B, C]
```

```
TailSet: [C, D, E]
```

Other Methods:

```
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(4);
        numbers.add(6);
        // numbers.add(3);
        System.out.println("TreeSet: " + numbers);

        // Using higher()
        System.out.println("Using higher: " + numbers.higher(4));

        // Using lower()
        System.out.println("Using lower: " + numbers.lower(3));

        // Using ceiling()
        System.out.println("Using ceiling: " + numbers.ceiling(4));

        // Using floor()
        System.out.println("Using floor: " + numbers.floor(3));

        System.out.println(numbers.contains(1));
        System.out.println(numbers.size());
```

```
System.out.println(numbers.isEmpty());
numbers.remove(3);
System.out.println("Set :" + numbers);

Object numbers2=numbers.clone();
System.out.println("Set :" + numbers2);

numbers.clear();
System.out.println("Set :" + numbers);

}
```

Output:

```
TreeSet: [2, 4, 5, 6]
```

```
Using higher: 5
```

```
Using lower: 2
```

```
Using ceiling: 4
```

```
Using floor: 2
```

```
false
```

```
4
```

```
false
```

```
Set :[2, 4, 5, 6]
```

```
Set :[2, 4, 5, 6]
```

```
Set :[]
```

As TreeSet implements Set interface, we can use TreeSet to perform various set operations like union, intersection etc..

Union of Sets:

To perform the union between two sets, we use the addAll() method

Example:

```
import java.util.TreeSet;;
```

```
class Main {
```

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

```
        TreeSet<Integer> evenNumbers = new TreeSet<>();
```

```
        evenNumbers.add(2);
```

```
        evenNumbers.add(4);
```

```
        System.out.println("TreeSet1: " + evenNumbers);
```

```
        TreeSet<Integer> numbers = new TreeSet<>();
```

```
        numbers.add(1);
```

```
        numbers.add(2);
```

```
        numbers.add(3);
```

```
        System.out.println("TreeSet2: " + numbers);
```

```
        // Union of two sets
```

```
        numbers.addAll(evenNumbers);
```

```
        System.out.println("Union is: " + numbers);
```

```
}
```

```
}
```

Output:

```
TreeSet1: [2, 4]
TreeSet2: [1, 2, 3]
Union is: [1, 2, 3, 4]
```

Intersection of sets:

To perform the intersection between two sets, we use the retainAll() method

Example:

```
import java.util.TreeSet;;
```

```
class Main {
```

```
    public static void main(String[] args) {
        TreeSet<Integer> evenNumbers = new TreeSet<>();
        evenNumbers.add(2);
        evenNumbers.add(4);
        System.out.println("TreeSet1: " + evenNumbers);
```

```
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        System.out.println("TreeSet2: " + numbers);
```

```
        // Intersection of two sets
        numbers.retainAll(evenNumbers);
        System.out.println("Intersection is: " + numbers);
    }
}
```

Output:

```
TreeSet1: [2, 4]
TreeSet2: [1, 2, 3]
Intersection is: [2]
```

Difference of Sets:

To calculate the difference between the two sets, we can use the `removeAll()` method.

Example:

```
import java.util.TreeSet;;
```

```
class Main {
```

```
    public static void main(String[] args) {
        TreeSet<Integer> evenNumbers = new TreeSet<>();
        evenNumbers.add(2);
        evenNumbers.add(4);
        System.out.println("TreeSet1: " + evenNumbers);
```

```
        TreeSet<Integer> numbers = new TreeSet<>();
```

```
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
```

```
        System.out.println("TreeSet2: " + numbers);
```

```
// Difference between two sets
```

```
        numbers.removeAll(evenNumbers);
```

```
        System.out.println("Difference is: " + numbers);
```

```
    }  
}  
}
```

Output:

```
TreeSet1: [2, 4]  
TreeSet2: [1, 2, 3, 4]  
Difference is: [1, 3]
```

Subset of a Set:

To check if a set is a subset of another set or not, we use the containsAll() method.

Example:

```
import java.util.TreeSet;
```

```
class Main {  
  
    public static void main(String[] args) {  
  
        TreeSet<Integer> numbers = new TreeSet<>();  
  
        numbers.add(1);  
  
        numbers.add(2);  
  
        numbers.add(3);  
  
        numbers.add(4);  
  
        System.out.println("TreeSet1: " + numbers);
```

```
TreeSet<Integer> primeNumbers = new TreeSet<>();  
  
primeNumbers.add(2);  
  
primeNumbers.add(3);  
  
System.out.println("TreeSet2: " + primeNumbers);
```

```
// Check if primeNumbers is subset of numbers
```

```
        boolean result = numbers.containsAll(primeNumbers);
        System.out.println("Is TreeSet2 subset of TreeSet1? " + result);
    }
}
```

Output:

```
TreeSet1: [1, 2, 3, 4]
TreeSet2: [2, 3]
Is TreeSet2 subset of TreeSet1? True
```

Conclusion:

- TreeSet is very much like HashSet (unique elements) and provides predictable iteration order (sorted). Sorted order can be overridden using custom comparator.
- TreeSet uses Red-Black tree internally. So the set could be thought as a dynamic search tree.
- When you need a structure which is operated read/write frequently and also should keep order, the TreeSet is a good choice.
- TreeSet is very useful collection class in cases where we want to handle duplicate records in sorted manner.

Reference links:

<https://www.geeksforgeeks.org/treeset-in-java-with-examples/>

<https://www.callicoder.com/java-treeset/>

<https://www.javatpoint.com/java-treeset>

<https://howtodoinjava.com/java/collections/java-treeset-class/>

<https://beginnersbook.com/2013/12/treeset-class-in-java-with-example/>

Upcasting and Downcasting in Java

Typecasting is converting one data type to another.

Up-casting – Converting a subclass type to a superclass type is known as up casting.

Down-casting – Converting a superclass type to a subclass type is known as downcasting.

Before we go into the details, suppose that we have the following class hierarchy:

Mammal > Animal > Dog, Cat

Mammal is the super interface:

```
1  public interface Mammal {  
2      public void eat();  
3  
4      public void move();  
5  
6      public void sleep();  
7  }
```

Animal is the abstract class:

```
1  public abstract class Animal implements Mammal {  
2      public void eat() {  
3          System.out.println("Eating...");  
4      }  
5  
6      public void move() {  
7          System.out.println("Moving...");  
8      }  
9  
10     public void sleep() {  
11         System.out.println("Sleeping...");  
12     }  
13 }
```

`Dog` and `Cat` are the two concrete sub classes:

```

1  public class Dog extends Animal {
2      public void bark() {
3          System.out.println("Gow gow!");
4      }
5      public void eat() {
6          System.out.println("Dog is eating...");
7      }
8  }
9
10 public class Cat extends Animal {
11     public void meow() {
12         System.out.println("Meow Meow!");
13     }
14 }
```

1. What is Upcasting in Java?

Upcasting is casting a subtype to a supertype, upward to the inheritance tree. Let's see an example:

```

1  Dog dog = new Dog();
2  Animal anim = (Animal) dog;
3  anim.eat();
```

Here, we cast the `Dog` type to the `Animal` type. Because `Animal` is the supertype of `Dog`, this casting is called upcasting.

Note that the actual object type does not change because of casting. The `Dog` object is still a `Dog` object. Only the reference type gets changed. Hence the above code produces the following output:

```
1  Dog is eating...
```

Upcasting is always safe, as we treat a type to a more general one. In the above example, an `Animal` has all behaviors of a `Dog`.

This is also another example of upcasting:

```
1  Mammal mam = new Cat();
```

```
2 Animal anim = new Dog();
```

- Why is Upcasting in Java?

Generally, upcasting is not necessary. However, we need upcasting when we want to write general code that deals with only the supertype. Consider the following class:

```
1 public class AnimalTrainer {  
2     public void teach(Animal anim) {  
3         anim.move();  
4         anim.eat();  
5     }  
6 }
```

Here, the `teach()` method can accept any object which is subtype of `Animal`. So objects of type `Dog` and `Cat` will be upcasted to `Animal` when they are passed into this method:

```
1 Dog dog = new Dog();  
2 Cat cat = new Cat();  
3  
4 AnimalTrainer trainer = new AnimalTrainer();  
5 trainer.teach(dog);  
6 trainer.teach(cat);
```

2. What is Downcasting in Java?

Downcasting is casting to a subtype, downward to the inheritance tree. Let's see an example:

```
1 Animal anim = new Cat();  
2 Cat cat = (Cat) anim;
```

Here, we cast the `Animal` type to the `Cat` type. As `Cat` is subclass of `Animal`, this casting is called downcasting.

Unlike upcasting, downcasting can fail if the actual object type is not the target object type. For example:

```
1 Animal anim = new Cat();  
2 Dog dog = (Dog) anim;
```

This will throw a `ClassCastException` because the actual object type is `Cat`. And a `Cat` is not a `Dog` so we cannot cast it to a `Dog`.

The Java language provides the `instanceof` keyword to check type of an object before casting. For example:

```
1  if (anim instanceof Cat) {  
2      Cat cat = (Cat) anim;  
3      cat.meow();  
4  } else if (anim instanceof Dog) {  
5      Dog dog = (Dog) anim;  
6      dog.bark();  
7  }
```

So if you are not sure about the original object type, use the `instanceof` operator to check the type before casting. This eliminates the risk of a `ClassCastException` thrown.

• Why is Downcasting in Java?

Downcasting is used more frequently than upcasting. Use downcasting when we want to access specific behaviors of a subtype.

Consider the following example:

```
1  public class AnimalTrainer {  
2      public void teach(Animal anim) {  
3          // do animal-things  
4          anim.move();  
5          anim.eat();  
6  
7          // if there's a dog, tell it barks  
8          if (anim instanceof Dog) {  
9              Dog dog = (Dog) anim;  
10             dog.bark();  
11         }  
12     }  
13 }
```

Here, in the `teach()` method, we check if there is an instance of a `Dog` object passed in, downcast it to the `Dog` type and invoke its specific method, `bark()`.

Okay, so far you have got the nuts and bolts of upcasting and downcasting in Java. Remember:

- Casting does not change the actual object type. Only the reference type gets changed.
- Upcasting is always safe and never fails.
- Downcasting can risk throwing a `ClassCastException`, so the `instanceof` operator is used to check type before casting.

Use parenthesis to override operator precedence

We can use the brackets or parenthesis to help us as they are evaluated first. We may look at a simple statement such as:

$2 + 3 * 5$

And as we read from left to right we may think the answer is 25; we would be wrong. The answer is 17. This is because the multiplication will happen before the addition. Just look at the M and A is BODMAS the M comes before A.

1. Brackets
2. Order Of (2 power of 3 etc)
3. Division
4. Multiplication
5. Addition
6. Subtraction

We could rewrite the statement so it would be come 25:

$(2 + 3) * 5$

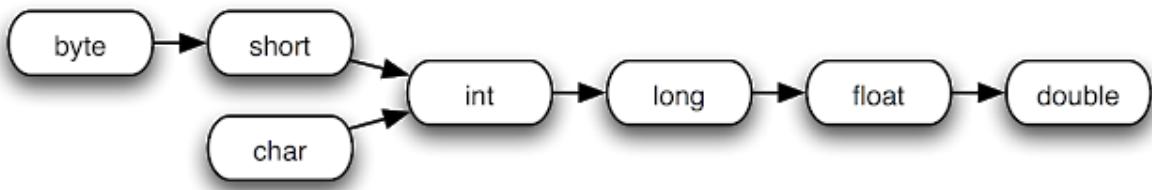
The contents of the brackets are evaluated first and then the multiplication. For the example we use in the JAVA OCA 1Z0-803 tutorial is to create a class to convert temperatures from Fahrenheit to Centigrade and vice – versa. To convert from Centigrade to Fahrenheit the formula is:

$((t * 9) / 5) + 32$

1. temperature $t * 9$
2. Divided by 5

3. and add 32

Java has similar rules when handle numeric primitives with different types -- the lower precision primitive types are promoted to higher precision primitive types before arithmetic calculating. Specifically byte and short are automatically promoted to int before any arithmetic calculations.



```
public class HelloWorld{  
    public static void main(String []args){  
        byte b = 1;  
        short s = 1;  
        short s1 = 1;  
        int i = 2;  
        long l = 1L;  
        long ll = 1; //auto cast int 1 to 1L  
        float f = 1.1f;  
        double d = 1.1;  
        long l1 = i - l; //i is converted from int to long  
        System.out.println(l1);  
        int i1 = b + s; //byte and short are automatically promoted to int  
        System.out.println(i1);  
        long l2 = b + l; //byte is automatically promoted to int, then to long when work with long.  
        System.out.println(l2);  
        double d1 = f * d; //float promote to double  
        System.out.println(d1);  
        double d2 = l + d; //long promote to double  
        System.out.println(d2);  
    }  
}
```

```

float f1 = l + f; //long promote to float
System.out.println(f1);

float f2 = b / f; //byte promote to float
System.out.println(f2);

float f3 = f % i; //int promote to float
System.out.println(f3);

double d3 = b++ * --f + s/b*d - (++i % 7.8f); //the highest precision type is double at right side
System.out.println(d3);

//short s3 = s + s1; //not compile
//short s4 = b + s; //not compile

}

}

```

Output:

```

1
2
2
1.2100000262260437
2.1
2.1
0.9090909
1.1
-2.899999976158142

```

- `++` and `--` take precedence when working with arithmetic operators.

```

class test{

public static void main(String[] a){

int i = 5;

System.out.println(i++ + 10/-i); //7
System.out.println(i); //5

}}

```

Explanation

```
i++ + 10/-i
equivalent to (i++) + 10/(--i)
execute i++ returns 5, i has value 6
5 + 10/(-i)
execute --i returns 5, i have value 5
5 + 10/5 = 5 + 2 = 7
```

- we can use `System.out.println(Long.MAX_VALUE)` to know the max value of the Long data type.
- Compound assignment operators (`+=`, `-=`, `*=`, `/=`) automatically do the type cast, so that we don't have to worry about compilation error due to type mismatch.

```
int i = 10;
long l = 10L;
//int i1 = i*l; //not compile
//int i1 = (int)i*l; //not compile
int i1 = (int)(i*l);
```

```
int i2 = 3;
//i2 = i2 + l; //not compile
i2 = (int)(i2 + l);
i2 += l; //equivalent to the previous line
```

- Assignment operator also returns the value of the assignment. For example, `(x = 5)` returns value 5.
Knowing this, we can assign `(x=5)` to another variable `y`.

```
class test{
    public static void main(String...args) {
        int x = 5;
        int y = (x = 5);
        System.out.println("x="+x+", y="+y);
```

```

int i;
int j;
System.out.println((j==((i=1)+3)+2));
}
}

OCAJP>javac test.java
OCAJP>java test
x=5, y=5
6

```

Java.util.Vector Class in Java

The Vector class implements a growable array of objects. Vectors basically fall in legacy classes but now it is fully compatible with collections.

- Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index
- They are very similar to ArrayList but Vector is synchronised and have some legacy method which collection framework does not contain.
- It extends **AbstractList** and implements **List** interfaces.

Constructor:

- **Vector():** Creates a default vector of initial capacity is 10.
- **Vector(int size):** Creates a vector whose initial capacity is specified by size.
- **Vector(int size, int incr):** Creates a vector whose initial capacity is specified by size and increment is specified by incr. It specifies the number of elements to allocate each time that a vector is resized upward.
- **Vector(Collection c):** Creates a vector that contains the elements of collection c.

Important points regarding Increment of vector capacity:

If increment is specified, Vector will expand according to it in each allocation cycle but if increment is not specified then vector's capacity get doubled in each allocation cycle. Vector defines three protected data member:

- **int capacityIncrement:** Contains the increment value.
- **int elementCount:** Number of elements currently in vector stored in it.
- **Object elementData[]:** Array that holds the vector is stored in it.

Methods in Vector:

1. **boolean add(Object obj)**: This method appends the specified element to the end of this vector.

Syntax: public boolean add(Object obj)

Returns: true if the specified element is added successfully into the Vector, otherwise it returns false.

Exception: NA.

```
// Java code illustrating add() method
import java.util.*;
class Vector_demo {
    public static void main(String[] args)
    {

        // create default vector
        Vector v = new Vector();

        v.add(1);
        v.add(2);
        v.add("geeks");
        v.add("forGeeks");
        v.add(3);

        System.out.println("Vector is " + v);
    }
}
```

Output:

```
[1, 2, geeks, forGeeks, 3]
```

2. **void add(int index, Object obj)**: This method inserts the specified element at the specified position in this Vector.

Syntax: public void add(int index, Object obj)

Returns: NA.

Exception: IndexOutOfBoundsException, method throws this exception

if the index (obj position) we are trying to access is out of range
(index < size()).

```
// Java code illustrating add() method
```

```
import java.util.*;  
  
class Vector_demo {  
  
    public static void main(String[] args)  
    {  
  
        // create default vector  
        Vector v = new Vector();  
  
        v.add(0, 1);  
        v.add(1, 2);  
        v.add(2, "geeks");  
        v.add(3, "forGeeks");  
        v.add(4, 3);  
  
        System.out.println("Vector is " + v);  
    }  
}
```

Output:

```
Vector is: [1, 2, geeks, forGeeks, 3]
```

3. **boolean addAll(Collection c)** This method appends all of the elements in the specified Collection to the end of this Vector.

```
Syntax: public boolean addAll(Collection c)
```

Returns: Returns true if operation succeeded otherwise false.

Exception: NullPointerException thrown if collection is null.

```
// Java code illustrating addAll()

import java.util.*;

class Vector_demo {

    public static void main(String[] arg)

    {

        ArrayList arr = new ArrayList();

        arr.add(3);

        arr.add("geeks");

        arr.add("forgeeks");

        arr.add(4);

        // create a default vector

        Vector v = new Vector();

        // copying all element of array list into vector

        v.addAll(arr);

        // checking vector v

        System.out.println("vector v:" + v);

    }

}
```

Output:

```
vector v:[3, geeks, forgeeks, 4]
```

4. **boolean addAll(int index, Collection c)** This method inserts all of the elements in the specified Collection into this Vector at the specified position.

Syntax: public boolean addAll(int index, Collection c)

Returns: true if this list changed as a result of the call.

Exception: IndexOutOfBoundsException -- If the index is out of range,

NullPointerException -- If the specified collection is null.

```
// Java code illustrating addAll()

import java.util.*;

class Vector_demo {

    public static void main(String[] arg)

    {

        ArrayList arr = new ArrayList();

        arr.add(3);

        arr.add("geeks");

        arr.add("forgeeks");

        arr.add(4);

        // createn default vector

        Vector v = new Vector();

        v.add(2);

        // copying all element of array list int0 vector

        v.addAll(1, arr);

        // checking vector v

        System.out.println("vector v:" + v);

    }

}
```

Output:

```
vector v:[2, 3, geeks, forgeeks, 4]
```

5. **void clear()** This method removes all of the elements from this vector.

Syntax: public void clear()

Returns: NA.

Exception: NA.

```
// Java code illustrating clear() method
import java.util.*;
class Vector_demo {
    public static void main(String[] arg)
    {

        // create default vector
        Vector v = new Vector();

        v.add(0, 1);
        v.add(1, 2);
        v.add(2, "geeks");
        v.add(3, "forGeeks");
        v.add(4, 3);

        System.out.println("Vector is: " + v);

        // clearing the vector
        v.clear();

        // checking vector
        System.out.println("after clearing: " + v);
    }
}
```

Output:

```
Vector is: [1, 2, geeks, forGeeks, 3]
```

```
after clearing: []
```

6. **Object clone()** This method returns a clone of this vector.

Syntax: public Object clone()

Returns: a clone of this ArrayList instance.

Exception: NA.

```
// Java code illustrating clone()  
import java.util.*;  
  
class Vector_demo {  
    public static void main(String[] arg)  
    {  
  
        // create default vector  
        Vector v = new Vector();  
  
        Vector v_clone = new Vector();  
  
        v.add(0, 1);  
        v.add(1, 2);  
        v.add(2, "geeks");  
        v.add(3, "forGeeks");  
        v.add(4, 3);  
  
        v_clone = (Vector)v.clone();  
  
        // checking vector  
        System.out.println("Clone of v: " + v_clone);  
    }  
}
```

```
}
```

Output:

```
Clone of v: [1, 2, geeks, forGeeks, 3]
```

7. **boolean contains(Object o)**: This method returns true if this vector contains the specified element.

Syntax: public boolean contains(object o)

Returns: true if the operation is succeeded otherwise false.

Exception: NA.

```
// Java code illustrating contains() method

import java.util.*;

class Vector_demo {

    public static void main(String[] arg)
    {

        // create default vector
        Vector v = new Vector();

        v.add(1);
        v.add(2);
        v.add("geeks");
        v.add("forGeeks");
        v.add(3);

        // check whether vector contains "forGeeks"
        if (v.contains("forGeeks"))

            System.out.println("forGeeks exists");

    }
}
```

Output:

forGeeks exists

8. **void ensureCapacity(int minCapacity):** This method increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument .

Syntax: public void ensureCapacity(int minCapacity)

Returns: NA.

Exception: NA.

```
// Java code illustrating ensureCapacity() method
import java.util.*;
class Vector_demo {
    public static void main(String[] arg)
    {

        // create default vector of capacity 10
        Vector v = new Vector();

        // ensuring capacity
        v.ensureCapacity(22);

        // cheking capacity
        System.out.println("Minimum capacity: " + v.capacity());
    }
}
```

Output:

Minimum capacity: 22

9. **Object get(int index):** This method returns the element at the specified position in this Vector.

Syntax: public Object get(int index)

Returns: returns the element at specified positions .

Exception: IndexOutOfBoundsException -- if the index is out of range.

```
// Java code illustrating get() methods

import java.util.*;

class Vector_demo {

    public static void main(String[] args)
    {

        // create default vector of capacity 10
        Vector v = new Vector();

        v.add(1);
        v.add(2);
        v.add("Geeks");
        v.add("forGeeks");
        v.add(4);

        // get the element at index 2
        System.out.println("element at indexx 2 is: " + v.get(2));
    }
}
```

Output:

```
element at indexx 2 is: Geeks
```

10. **int indexOf(Object o):** This method returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.

Syntax: public int indexOf(Object o)

Returns: the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

Exception: NA.

```
// Java code illustrating indexOf() method

import java.util.*;

class Vector_demo {

    public static void main(String[] arg)

    {

        // create default vector of capacity 10
        Vector v = new Vector();

        v.add(1);
        v.add(2);
        v.add("Geeks");
        v.add("forGeeks");
        v.add(4);

        // get the element at index of Geeks
        System.out.println("index of Geeks is: " + v.indexOf("Geeks"));

    }

}
```

Output:

```
index of Geeks is: 2
```

11. boolean isEmpty(): This method tests if this vector has no components.

Syntax: public boolean isEmpty()

Returns: true if vector is empty otherwise false.

Exception: NA.

```
// Java code illustrating isEmpty() method

import java.util.*;

class Vector_demo {

    public static void main(String[] arg)

    {

        // create default vector of capacity 10
        Vector v = new Vector();

        v.add(1);
        v.add(2);
        v.add("Geeks");
        v.add("forGeeks");
        v.add(4);

        v.clear();

        // check whether vector is empty or not
        if (v.isEmpty())
            System.out.println("Vector is clear");
    }
}
```

Output:

```
Vector is clear
```

12. int lastIndexOf(Object o): This method returns the index of the last occurrence of the specified element in this vector, or -1 if this vector does not contain the element.

Syntax: public int lastIndexOf(Object o)

Returns: returns the index of the last occurrence of the

specified element in this list, or -1 if this list does not contain the element.

Exception: NA.

```
// Java code illustrating lastIndexof()  
import java.util.*;  
  
class Vector_demo {  
  
    public static void main(String[] arg)  
    {  
  
        // create default vector of capacity 10  
        Vector v = new Vector();  
  
        v.add(1);  
        v.add(2);  
        v.add("Geeks");  
        v.add("forGeeks");  
        v.add(4);  
  
        // checking last occurrence of 2  
        System.out.println("last occurrence of 2 is: " + v.lastIndexOf(2));  
    }  
}
```

Output:

```
last occurrence of 2 is: 1
```

- 13. boolean remove(Object o):** This method removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.

Syntax: public boolean remove(Object o)

Returns: Returns the first occurrence of element.

Exception: NA.

```

// Java code illustrating remove method()

import java.util.*;

class Vector_demo {

    public static void main(String[] arg)

    {

        // create default vector of capacity 10
        Vector v = new Vector();

        v.add(1);
        v.add(2);
        v.add("Geeks");
        v.add("forGeeks");
        v.add(4);

        // removing first occurrence element at 1
        v.remove(1);

        // checking vector
        System.out.println("after removal: " + v);
    }
}

```

Output:

```
after removal: [1, Geeks, forGeeks, 4]
```

- 14. boolean equals(Object o):** This method compares the specified Object with this Vector for equality.

Syntax: public boolean equal(Object o)

Returns: true if operation succeeded otherwise false.

Exception: NA.

```
// Java code illustrating equals() method

import java.util.*;

class Vector_demo {

    public static void main(String[] arg)

    {

        // create default vector of capacity 10
        Vector v = new Vector();

        v.add(1);
        v.add(2);
        v.add("Geeks");
        v.add("forGeeks");
        v.add(4);

        // second vector
        Vector v_2nd = new Vector();

        v_2nd.add(1);
        v_2nd.add(2);
        v_2nd.add("Geeks");
        v_2nd.add("forGeeks");
        v_2nd.add(4);

        if (v.equals(v_2nd))
            System.out.println("both vectors are equal");
    }
}
```

Output:

both vectors are equal

15. Object firstElement(): This method returns the first component (the item at index 0) of this vector.

Syntax: public Object firstElement()

Returns: NA.

Exception: NoSuchElementException -- This exception is returned if this vector has no components.

```
// Java code illustrating firstElement() method
import java.util.*;
class Vector_demo {
    public static void main(String[] args)
    {

        // create default vector of capacity 10
        Vector v = new Vector();

        v.add(1);
        v.add(2);
        v.add("Geeks");
        v.add("forGeeks");
        v.add(4);

        // first element of vector
        System.out.println("first element of vector is: " + v.firstElement());
    }
}
```

Output:

first element of vector is: 1

16. void trimToSize(): This method trims the capacity of this vector to be the vector's current size.

Syntax: public void trimToSize()

Returns: NA.

Exception: NA.

```
// Java code illustrating trimToSize() method

import java.util.*;

class Vector_demo {

    public static void main(String[] arg)

    {

        // create default vector of capacity 10
        Vector v = new Vector();

        v.add(1);
        v.add(2);
        v.add("Geeks");
        v.add("forGeeks");
        v.add(4);

        // checking initial capacity
        System.out.println("Initial capacity: " + v.capacity());

        // trim capacity to size
        v.trimToSize();

        // checking capacity after trimming
        System.out.println("capacity after trimming: " + v.capacity());
```

```
    }  
}
```

Output:

```
Initial capacity: 10  
capacity after triming: 5
```

17. String `toString()`: The `toString()` method is used to return a string representation of this Vector, containing the String representation of each element.

Syntax: `public String toString()`

Returns: a string representation of this collection.

Exception: NA

```
// Java code illustrating toString() method  
  
import java.util.*;  
  
class Vector_demo {  
  
    public static void main(String[] args)  
    {  
  
        // create default vector of capacity 10  
        Vector v = new Vector();  
  
        v.add(1);  
        v.add(2);  
        v.add("Geeks");  
        v.add("forGeeks");  
        v.add(4);  
  
        // string equivalent of vector  
        System.out.println(" String equivalent of vector: " + v.toString());  
    }  
}
```

```
}
```

Output:

```
String equivalent of vector: [1, 2, Geeks, forGeeks, 4]
```

18. `object[] toArray()`: This method returns a array representation of this Vector, containing the String representation of each element.

Syntax: `public object[] toArray()`

Returns: an array containing all of the elements in this collection.

Exception: NA.

```
// Java code illustrating toArray() method
import java.util.*;
class Vector_demo {
    public static void main(String[] arg)
    {

        String elements[] = { "M", "N", "O", "P", "Q" };
        Set set = new HashSet(Arrays.asList(elements));

        String[] strObj = new String[set.size()];
        strObj = (String[])set.toArray(strObj);

        for (int i = 0; i < strObj.length; i++) {
            System.out.println(strObj[i]);
        }
        System.out.println(set);
    }
}
```

```
    }  
}
```

Output:

```
P  
Q  
M  
N  
O  
[P, Q, M, N, O]
```

19. **int size():** This method returns the number of components in this vector.

Syntax: public int size()

Returns: returns the number of components in this vector.

Exception: NA

```
// Java code illustrating size() method  
import java.util.*;  
  
class Vector_demo {  
  
    public static void main(String[] arg)  
    {  
  
        // create default vector of capacity 10  
        Vector v = new Vector();  
  
        v.add(1);  
        v.add(2);  
        v.add("Geeks");  
        v.add("forGeeks");  
        v.add(4);
```

```
// size of vector  
System.out.println(" size of vector: " + v.size());  
}  
}
```

Output:

```
size of vector: 5
```

20. void setSize(int newSize): This method sets the size.

Syntax: public void setSize(int newSize)

Returns: NA.

Exception: ArrayIndexOutOfBoundsException -- This exception is thrown if the new size is negative.

```
// Java code illustrating setSize() method  
import java.util.*;  
  
class Vector_demo {  
  
    public static void main(String[] arg)  
    {  
  
        // create default vector of capacity 10  
        Vector v = new Vector();  
  
        v.add(1);  
        v.add(2);  
        v.add("Geeks");  
        v.add("forGeeks");  
        v.add(4);  
  
        // setting new size of vector  
        v.setSize(13);
```

```
// size of vector  
System.out.println("size of vector: " + v.size());  
}  
}
```

Output:

```
size of vector: 13
```

21. void setElementAt(Object obj, int index): This method sets the component at the specified index of this vector to be the specified object.

Syntax: public void setElementAt(E obj, int index)

Returns: NA.

Exception: ArrayIndexOutOfBoundsException -- This exception is thrown

if the accessed index is out of range.

```
// Java code illustrating setElementAt() method  
import java.util.*;  
class Vector_demo {  
    public static void main(String[] args)  
    {  
        // create default vector of capacity 10  
        Vector v = new Vector();  
  
        v.add(1);  
        v.add(2);  
        v.add("Geeks");  
        v.add("forGeeks");  
        v.add(4);  
  
        // set 4 at the place of 2
```

```
v.setElementAt(4, 1);

System.out.println("vector: " + v);
}

}
```

Output:

```
vector: [1, 4, Geeks, forGeeks, 4]
```

22. retainAll(Collection c): This method retains only the elements in this Vector that are contained in the specified Collection.

Syntax: public boolean retainAll(Collection c)

Returns: true if this Vector is changed as a result of the call.

Exception: NullPointerException -- This exception is thrown if the

specified collection is null.

```
// Java code illustrating retainAll() method

import java.util.*;
class Vector_demo {

    public static void main(String[] arg)
    {

        Vector vec = new Vector(7);

        Vector vecretain = new Vector(4);

        // use add() method to add elements in the vector
        vec.add(1);
        vec.add(2);
        vec.add(3);
        vec.add(4);
        vec.add(5);
        vec.add(6);
        vec.add(7);
```

```

// this elements will be retained
vecretain.add(5);
vecretain.add(3);
vecretain.add(2);

System.out.println("Calling retainAll()");
vec.retainAll(vecretain);

// let us print all the elements available in vector
System.out.println("Numbers after removal :- ");

Iterator itr = vec.iterator();

while (itr.hasNext()) {
    System.out.println(itr.next());
}
}

```

Output:

```

Calling retainAll()

Numbers after removal :- 

2
3
5

```

23. void removeAllElements(): This method removes all components from this vector and sets its size to zero.

Syntax: public void removeAllElements()

Returns: NA.

Exception: NA.

```
// Java code illustrating removeAllElements() method

import java.util.*;

class Vector_demo {

    public static void main(String[] arg)

    {

        Vector vec = new Vector(7);

        // use add() method to add elements in the vector

        vec.add(1);

        vec.add(2);

        vec.add(3);

        vec.add(4);

        vec.add(5);

        vec.add(6);

        vec.add(7);

        // remove all elements

        vec.removeAllElements();

        // checking vector's size

        System.out.println("Size: " + vec.size());

        // checking vector's components

        System.out.println("vector's components: " + vec);

    }

}
```

Output:

```
Size: 0

vector's components: []
```

24. Object lastElement(): This method returns the last component of the vector.

Syntax: public Object lastElement()

Returns: returns the last component of the vector,
i.e., the component at index size() - 1.

Exception: NoSuchElementException -- This exception is thrown
if this vector is empty

```
// Java code illustrating lastElement() method

import java.util.*;

class Vector_demo {

    public static void main(String[] args)
    {
        Vector vec = new Vector(7);

        // use add() method to add elements in the vector
        vec.add(1);
        vec.add(2);
        vec.add(3);
        vec.add(4);
        vec.add(5);
        vec.add(6);
        vec.add(7);

        // checking last element of vector
        System.out.println("vector's last components: " + vec.lastElement());
    }
}
```

Output:

```
vector's last components: 7
```

25. int hashCode(): This method returns the hash code value for this Vector.

Syntax: public int hashCode()

Returns: returns the hash code value(int) for this list.

Exception: NA.

```
// Java code illustrating hashCode() method
import java.util.*;
class Vector_demo {
    public static void main(String[] arg)
    {
        Vector vec = new Vector(7);

        // use add() method to add elements in the vector
        vec.add(1);
        vec.add(2);
        vec.add(3);
        vec.add(4);
        vec.add(5);
        vec.add(6);
        vec.add(7);

        // checking hash code
        System.out.println("Hash code: " + vec.hashCode());
    }
}
```

Output:

```
Hash code: -1604500253
```

26. boolean removeElement(Object obj): This method removes the first occurrence of the argument from this vector.

Syntax: public boolean removeElement(Object obj)

Returns: true if operation is succeeded otherwise false.

Exception:

```
// Java code illustrating removeElement()

import java.util.*;

class Vector_demo {

    public static void main(String[] args)
    {
        Vector vec = new Vector(7);

        // use add() method to add elements in the vector
        vec.add(1);
        vec.add(2);
        vec.add(3);
        vec.add(4);
        vec.add(5);
        vec.add(6);
        vec.add(7);

        // remove an element
        vec.removeElement(5);

        // checking vector
        System.out.println("Vector after removal: " + vec);
    }
}
```

Output:

```
Vector after removal: [1, 2, 3, 4, 6, 7]
```

27. void copyInto(Object[] anArray):This method copies the components of this vector into the specified array.

Syntax: public void copyInto(Object[] anArray)

Returns: NA.

Exception: NullPointerException -- if the given array is null.

```
// Java code illustrating copyInto() method

import java.util.*;

class Vector_demo {

    public static void main(String[] arg)
    {
        Vector vec = new Vector(7);

        // use add() method to add elements in the vector
        vec.add(1);
        vec.add(2);
        vec.add(3);
        vec.add(4);
        vec.add(5);
        vec.add(6);
        vec.add(7);

        Integer[] arr = new Integer[7];

        // copy component of vector int array arr
        vec.copyInto(arr);

        System.out.println("elements in array arr: ");
        for (Integer num : arr) {
            System.out.println(num);
        }
    }
}
```

```
    }  
}  
}
```

Output:

```
elements in array arr:  
1  
2  
3  
4  
5  
6  
7
```

28. **int capacity()**: This method returns the current capacity of this vector.

Syntax: public int capacity()

returns: returns the current capacity of the vector as an integer value. Here capacity means the length of its internal data array, kept in the field elementData of this vector.

Exception: NA.

```
// Java code illustrating capacity() method  
import java.util.*;  
  
class Vector_demo {  
  
    public static void main(String[] arg)  
    {  
        Vector vec = new Vector(7);  
  
        // use add() method to add elements in the vector
```

```

        vec.add(1);

        vec.add(2);

        vec.add(3);

        vec.add(4);

        vec.add(5);

        vec.add(6);

        vec.add(7);

    // checking capacity
    System.out.println("Capacity of vector: " + vec.capacity());
}

}

```

Output:

```
Capacity of vector: 7
```

29. void insertElementAt(Object obj, int index): This method inserts the specified object as a component in this vector at the specified index.

Syntax: public void insertElementAt(E obj, int index)

Returns: NA.

Exception: ArrayIndexOutOfBoundsException -- This exception is thrown

if the index is invalid.

```

// Java code illustrating insertElementAt() method

import java.util.*;

class Vector_demo {

    public static void main(String[] args)

    {

        Vector vec = new Vector(7);

        // use add() method to add elements in the vector

        vec.add(1);

```

```

    vec.add(2);

    vec.add(3);

    vec.add(4);

    vec.add(5);

    vec.add(6);

    vec.add(7);

    // insert 10 at the index 7
    vec.insertElementAt(10, 7);

    // checking vector
    System.out.println(" Vector: " + vec);
}

}

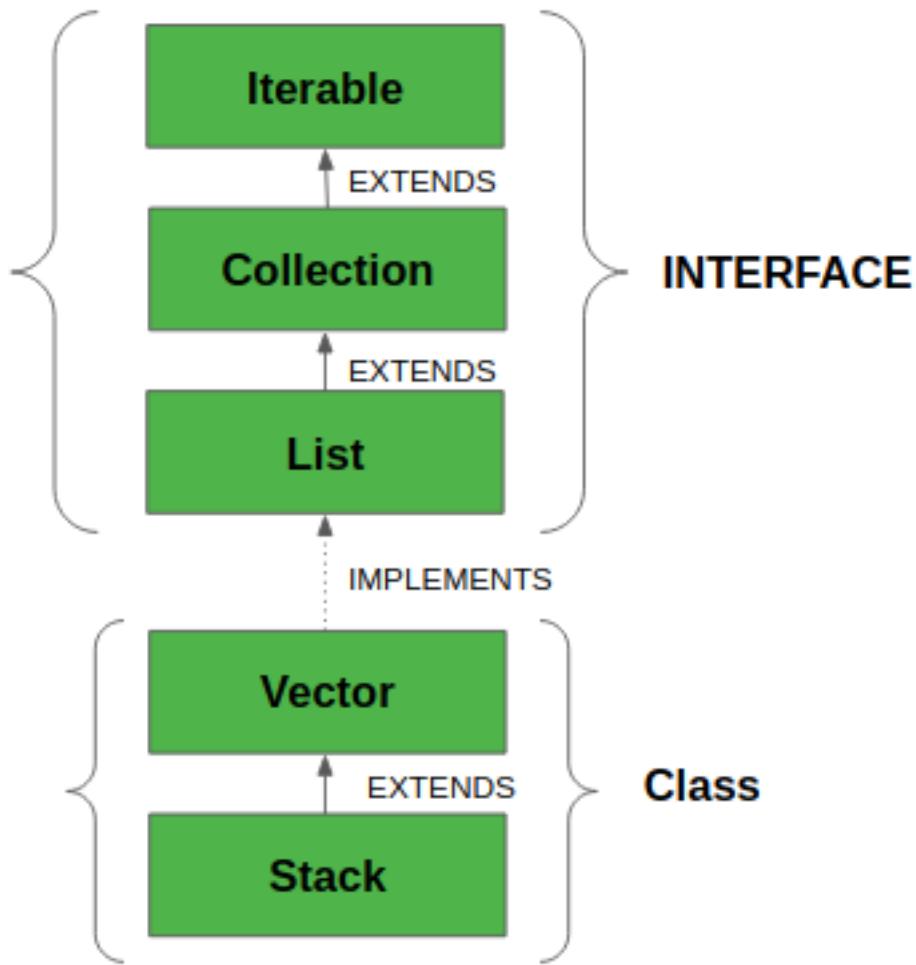
```

Output:

Vector: [1, 2, 3, 4, 5, 6, 7, 10]

Stack Class in Java

Java Collection framework provides a Stack class which models and implements Stack data structure. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek. The class can also be said to extend Vector and treats the class as a stack with the five mentioned functions. The class can also be referred to as the subclass of Vector. This diagram shows the hierarchy of Stack class:



The class supports one *default constructor* **Stack()** which is used to *create an empty stack*.

Below program shows few basic operations provided by the Stack class:

```

// Java code for stack implementation

import java.io.*;

```

```
import java.util.*;  
  
class Test  
{  
  
    // Pushing element on the top of the stack  
    static void stack_push(Stack<Integer> stack)  
    {  
  
        for(int i = 0; i < 5; i++)  
        {  
  
            stack.push(i);  
        }  
  
    }  
  
    // Popping element from the top of the stack  
    static void stack_pop(Stack<Integer> stack)  
    {  
  
        System.out.println("Pop :");  
  
        for(int i = 0; i < 5; i++)  
        {  
  
            Integer y = (Integer) stack.pop();  
            System.out.println(y);  
        }  
  
    }  
  
    // Displaying element on the top of the stack  
    static void stack_peek(Stack<Integer> stack)  
    {  
  
        Integer element = (Integer) stack.peek();  
        System.out.println("Element on stack top : " + element);  
    }  
}
```

```

// Searching element in the stack

static void stack_search(Stack<Integer> stack, int element)
{
    Integer pos = (Integer) stack.search(element);

    if(pos == -1)
        System.out.println("Element not found");
    else
        System.out.println("Element is found at position " + pos);
}

```

```

public static void main (String[] args)
{
    Stack<Integer> stack = new Stack<Integer>();

    stack_push(stack);
    stack_pop(stack);
    stack_push(stack);
    stack_peek(stack);
    stack_search(stack, 2);
    stack_search(stack, 6);
}

}

```

Output:

Pop :

4
3
2
1

```
0
```

```
Element on stack top : 4
```

```
Element is found at position 3
```

```
Element not found
```

Methods in Stack class

1. **Object push(Object element)** : Pushes an element on the top of the stack.
2. **Object pop()** : Removes and returns the top element of the stack. An 'EmptyStackException' exception is thrown if we call pop() when the invoking stack is empty.
3. **Object peek()** : Returns the element on the top of the stack, but does not remove it.
4. **boolean empty()** : It returns true if nothing is on the top of the stack. Else, returns false.
5. **int search(Object element)** : It determines whether an object exists in the stack. If the element is found, it returns the position of the element from the top of the stack. Else, it returns -1.