Core JAVA Left over Topics-Part 1(Main Part - 7)

Object class methods

```
public class Object {
    //Consturctor of the Class
    public Object();

    //To know the name of the class which implements an interface at the runtime.
    public final native Class<?> getClass();

    //At the time of creating of Object,JVM will mantain unique address.
    public native int hashCode();

    //To compare the objects
    public boolean equals(java.lang.Object);

    //It is used during cloning of Object
    protected native Object clone() throws CloneNotSupportedException;

    //While printing the reference the method gets called automatically
    public String toString();

    //methods used while working with MultiThreading Environment[Interthread communication]
    public final native void notify();
    public final native void notifyAll();
    public final void wait() throws InterruptedException;
    public final native void wait(long) throws InterruptedException;
    public final void wait(long, int) throws InterruptedException;

    //Called by Garbage Collector beforing cleaning the object from the Heap memory.
    protected void finalize() throws Throwable;
}
```

A software is installed to check the predefined code of .class files (20-30 min)

toString()

=> We can use this method to get the String representation of the Object.

=> When we try to print the reference,this method gets called automatically(Magic method).

=> if our class doesn't contain this method, then Object class toString() will be called automatically.

Note::

```
class Object
{
    public String toString() {
        //method body
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }
}
```

eg#1.

Before Overriding toString() of Object class

```
class Student
{
    String name;
    int rollNo;
    Student(String name,int rollNo)
    {
        this.name = name;
        this.rollNo = rollNo;
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Student s1 = new Student("sachin",10);
        Student s2 = new Student("dhoni",7);
        System.out.println(s1);
        System.out.println(s2);
        System.out.println();
        System.out.println(s1.toString());
        System.out.println(s2.toString());
    }
}
```

Output

Student@15db9742

Student@6d06d69c

Student@15db9742

Student@6d06d69c

After Overriding toString() in Student class

```
class Student
{
    String name;
    int rollNo;
    Student(String name,int rollNo)
    {
        this.name = name;
        this.rollNo = rollNo;
    }
    @Override
    public String toString()
    {
        return name +"-----" + rollNo;
    }
}
```

Output

sachin-----10

dhoni-----7

sachin-----10

dhoni-----7

Can u name few inbuilt classes where toString() is overriden to print the Object data than the Object reference?

Ans. String,StringBuilder,StringBuffer,Wrapper classes,Thread class ,All Collection objects.....

Class<?> getClass();

This method is used to get the runtime object details while dealing with "Implementation classes".

eg#1.

```
interface INotification
{
    public void notifyUsers();
}
```

```
class EmailNotification implements INotification
{
    @Override
    public void notifyUsers(){
        //logic to notify the user through email...
        System.out.println("Notification through EMAIL....");
    }
}
class SMSNotification implements INotification
{
    @Override
    public void notifyUsers(){
        //logic to notify the user through sms...
        System.out.println("Notification through SMS....");
    }
}
class Factory{
    public INotification getInstance(String mode){
        INotification notification =null;
        if (mode.equalsIgnoreCase("SMS"))
            notification = new SMSNotification();
        else if(mode.equalsIgnoreCase("EMAIL"))
            notification = new EmailNotification();
        else if(mode.equalsIgnoreCase("PUSH"))
            notification =new PushNotification();
        else
            throw new RuntimeException("Mode is invalid...");
        return notification;
    }
}
class PushNotification implements INotification
{
    //loose coupling,abstraction
    @Override
    public void notifyUsers(){
        //logic to notify the user through push service...
        System.out.println("Notification through PUSH ....");
```

```
    }
}
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        Factory factory= new Factory();
        //loose coupling,abstraction
        INotification obj1 = factory.getInstance("SMS");
        obj1.notifyUsers();
        System.out.println("Notification object is :: "+obj1.getClass());

        System.out.println();

        //loose coupling,abstraction
        INotification obj2 = factory.getInstance("EMAIL");
        obj2.notifyUsers();
        System.out.println("Notification object is :: "+obj2.getClass());
        System.out.println();
        INotification obj3 = factory.getInstance("PUSH");
        obj3.notifyUsers();
        System.out.println("Notification object is :: "+obj3.getClass());

        System.out.println();

        INotification obj4 = factory.getInstance("Call");
        obj4.notifyUsers();
        System.out.println("Notification object is :: "+obj4.getClass());
    }
}
```

Output

Notification through SMS....

Notification object is :: class SMSNotification

Notification through EMAIL....

Notification object is :: class EmailNotification

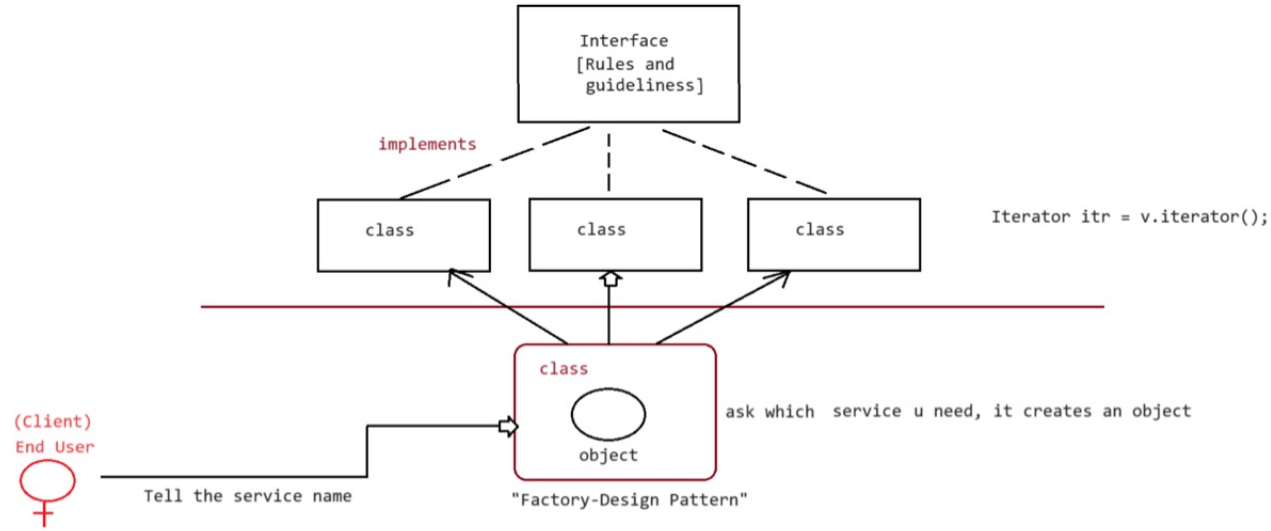
Notification through PUSH ....

Notification object is :: class PushNotification

Exception in thread "main" java.lang.RuntimeException: Mode is invalid...

at Factory.getInstance(Test.java:36)

at Test.main(Test.java:79)



"Solution to a realtime problem faced by developers is given by Designpattern".
Design Patterns are standard solution,if we implement it in our project then the problem won't occur for our clients.
Implementing the Design pattern is a must and knowledge of Design Pattern is a must for Developer.

Main idea for above diagram and program is to show that behind calling getClass a lot of operations are performed like in above program instead of making objects of different notifications in main we created it in separate class(Factory) and then call its method. With this if we need to add new notification then we need to change Factory class instead of main.It also shows loose coupling + abstraction.It is same in Iterator as well, we can use iterator with any data structure so internally it has to handle all those types.

hashcode()

public int hashCode()

=> For every object jvm will generate a unique number which is nothing but a hashCode.

=> For all hashing related datastructure like HashSet,HashMap,...jvm will use hashcode to save the objects.

=> If objects are stored according to the hashcode searching for a data becomes more efficient.

Note:if we don't override hashCode(),then object class hashCode() will be executed which generates the hashCode based on the address of the object.

=> Overriding the hashCode() is said to be proper, iff for every object we have to generate a unique number as a hashcode for every object.

```
class Object
{
    public String toString(){
```

```
        return getClass().getName() + "@"+ Integer.toHexString(hashCode());
    }
    //body will come at the runtime:: by linking with other language libraries
    public native int hashCode(); // object address(memory level hence will not be from java, c language will be used for such cases) -> hashcode

}
```

eg#1.

```
class Student
{
    @Override
    public int hashCode()
    {
        return 10;
    }
}
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        Student s1 = new Student();
        System.out.println(s1.hashCode());
        Student s2 = new Student();
        System.out.println(s2.hashCode());
    }
}
```

Hashcode for 2 objects should not be same it is a improper approach.

Output

10

10

Sol →

eg#2.

```
class Student
{
```

```
    int rollNo;
    Student(int rollNo){
        this.rollNo = rollNo;
    }
    @Override
    public int hashCode()
    {
        return rollNo;
    }
}
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        Student s1 = new Student(10);
        System.out.println(s1.hashCode());
        Student s2 = new Student(7);
        System.out.println(s2.hashCode());
    }
}
```

Output

10

7

1)- => Which class toString() and hashCode() will be called?

Ans.toString() :: Object class

hashCode() :: Object class

eg#1.

```
class Student
{
    int rollNo;
    Student(int rollNo){
        this.rollNo = rollNo;
    }
}
```

```
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        Student s1 = new Student(10);
        System.out.println(s1);
        Student s2 = new Student(7);
        System.out.println(s2);
    }
}
```

2)- => Which class toString() and hashCode() will be called?

Ans.toString() :: Object class

hashCode() :: Student class

eg#1.

```
class Student
{
    int rollNo;
    Student(int rollNo){
        this.rollNo = rollNo;
    }
    @Override
    public int hashCode(){
        return rollNo;
    }
}
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        Student s1 = new Student(10);
        System.out.println(s1);
        Student s2 = new Student(7);
        System.out.println(s2);
    }
```

```
}
```

Output

Student@a

Student@7

eg#3.

```
class Student
{
    int rollNo;
    String name;
    Student(int rollNo,String name){
        this.rollNo = rollNo;
        this.name = name;
    }
    @Override
    public int hashCode(){
        return rollNo;
    }
    @Override
    public String toString(){
        return name + " "+ rollNo;
    }
}
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        Student s1 = new Student(10,"sachin");
        System.out.println(s1);
        Student s2 = new Student(7,"dhoni");
        System.out.println(s2);
    }
}
```

Output

sachin 10

dhoni 7

Note: IMP POINTS →

1. **If we are overriding toString(), then making a call to hashCode() is in the hands of programmer, hashCode() won't be called automatically.**
2. **if we don't override toString(), then toString() internally will make a call to hashCode(), so overriding toString() and hashCode() is totally decided by the programmer.**

equals()

=> To check whether two objects are equal or not we use this method.

=> If our class doesn't contain equals() then object class equals() will be executed.

```
public boolean equals(Object obj){
    return (this == obj);  //reference check
}
```

eg#1.

```
class Student
{
    int rollNo;
    String name;
    Student(int rollNo,String name){
        this.rollNo = rollNo;
        this.name = name;
    }
}
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        Student s1 = new Student(10,"sachin");
        Student s2 = new Student(7,"dhoni");
        Student s3 = new Student(10,"sachin");
        Student s4 = s1;
        System.out.println(s1.equals(s2));
        System.out.println(s1.equals(s3));
        System.out.println(s1.equals(s4));
    }
```

}

while overridng equals() we need to make sure, we handle the following cases

1. Compare the entire object data, based on the result return true or false.
2. if we are passing different type of data, it would result in "**ClassCastException**"
   a. handle the ClassCastException and return false.
3. if we are passing null type of data, it would result in "**NullPointerException**"
   a. handle the NullPointerException and return false.

eg#1. → here we are making equals as per our requirement(not as per object)

```
class Student
{
    int rollNo;
    String name;
    Student(int rollNo,String name){
        this.rollNo = rollNo;
        this.name = name;
    }
    //overriding equals() as per our requirement
    @Override
    public boolean equals(Object obj)
    {
        try
        {
            //compare the contents and return boolean result
            int id1= this.rollNo;
            String name1 = this.name;
            Student std = (Student)obj;
            int id2 = std.rollNo;
            String name2 = std.name;
            if (id1==id2 && name1.equals(name2)) // String class equals method will be called for name(name1.equals(name2)
            {
                return true;
            }
            else
            {
                return false;
            }
```

```
            }
            catch (ClassCastException ce)
            {
                //write the logic
                return false;
            }
            catch(NullPointerException ne)
            {
                //write the logic
                return false;
            }
            catch(Exception e)
            {
                //write the logic
                return false;
            }
        }
}
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        Student s1 = new Student(10,"sachin");
        Student s2 = new Student(7,"dhoni");
        Student s3 = new Student(10,"sachin");
        Student s4 = s1;
        System.out.println(s1.equals(s2));//false
        System.out.println(s1.equals(s3));//true
        System.out.println(s1.equals(s4));//true
        System.out.println(s1.equals("divya"));
        System.out.println(s1.equals(null));
    }
}
```

Output

false

true

true

false

false

More simplified version of equals() →

1)- just using rollNo instead of this.rollNo(as it will look for current object only)

2)- removing else block and returning false at the end

```
public boolean equals(Object obj) {
    try
    {
        Student std = (Student)obj;
        //compare the contents and return boolean result
        if (rollNo==std.rollNo && name.equals(std.name))
            return true;
    }
    catch (ClassCastException ce)
    {
        //write the logic
        return false;
    }
    catch(NullPointerException ne)
    {
        //write the logic
        return false;
    }
    catch(Exception e)
    {
        //write the logic
        return false;
    }
    return false;
}
```

More simplified version of equals()

1)- Compare the reference if they are same return true.

2)- Exception will only come when the object is not of the type of Student. Hence compare only when object is of Student instance.

```
public boolean equals(Object obj) {
```

```
    if (this == obj)
    {
        return true;
    }
    if (obj instanceof Student)
    {
        Student std = (Student)obj;
        //compare the contents and return boolean result
        if (rollNo==std.rollNo && name.equals(std.name))
        return true;
    }
    return false;
}
```

Key interface of Collection

a. Collection

b. List => ArrayList, LinkedList, Stack,Vector

=> Duplicates are allowed, insertion order is preserved,null is also allowed.

=> Accessing the element is based on index

=> interfaces it implements are "Serializable,Cloneable,RandomAccess".

=> ArrayList,Vector best suitable when we perform read operation,where as linkedlist best suited when we perform insert/delete operation.

c. Set => HashSet,LinkedHashSet

=> Duplicates are not allowed, insertion order is not preserved due to hashing.

=> interfaces it implements are "Serializable,Cloneable".

=> If we want Elements to be in inserted order then we need to go for"LinkedHash Set".

d. NavigableSet

e. SortedSet => TreeSet

=> Duplicates are not allowed, insertion is not preserved due to hashing

=> null is not allowed it would result in "NullPointerException".

=> The elements are arranged in sorted order.

=> if the elements are of homogenous type, then by default Comparable logic is used for

Sorting the elements

=> if the elements are of heteregenous type, then it would result in "ClassCastException".

=> To sort the hetergenous type of objects we need to use "Comparator" interface.

f. Map => HashMap,WeakHashMap,LinkedHashMap,IdentityHashMap

=> It represents the data in the form of "<K,V>" pair.

=> Keys can't be duplicate whereas the values can be duplicated.

=> WeakHashMap vs HashMap

a. if the key is null and if it is a part of HashMap then GC can't clean the object.

b. if the key is null and if it is a part of WeakHashMap then GC can clean the object.

=> IdentityHashMap vs HashMap

a. To identify duplicate key, JVM will use "==" operator in case of IdentityHashMap.

b. To identify duplicate key, JVM will use equals() in case of HashMap.

g. NavigableMap

h. SortedMap => TreeMap(K,V)

=> Duplicates keys are not allowed, insertion is not preserved due to hashing.

=> null is not allowed as a key, it would result in "NullPointerException".

=> The elements are arranged in sorted order.

=> if the value are of homogenous type, then by default Comparable logic is used for Sorting the elements

=> if the values are of heteregenous type, then it would result in "ClassCastException".

=> To sort the hetergenous type of objects we need to use "Comparator" interface.

i. Queue =>

PriorityQueue,BlockingQueue,PriorityBlockingQueue,LinkedBlockingQueue

=> Prior to processing, if we want to represent a group of individual objects we go for Queue.

=> it follows the order of "FIFO".

Cursors of Collection

1. **Enumeration =>** Legacy cursor applicable only for Legacy classes. Operations applicable are :: read
2. **Iterator =>** Universal Cursor applicable for any collection object.
   Operations applicable are :: read[only forward],remove
3. **ListIterator =>** Cursor applicable only for List objects.
   Operations applicable are ::
   read[forward,backward],remove,update,insert,delete

Sorting based interfaces

a. Comparable :: Default Natural Sorting Order.
   public int compareTo(Object obj)

b. Comparator :: Customized Sorting Order.
   public int compare(Object obj1,Object obj2)

1.5 version enhancement of Queue

1. It is a child interface of Collection
2. If we want to represent a group of individual Objects before Processing then we should go for Queue.
3. From 1.5 version LinkedList also implements Queue
4. Usually Queue follows FIFO Order,Based on our requirement we can implement our own priority also.
5. LinkedList based implementation Queue also follows FIFO order.

eg: Before sending mail, we need to store the mail id in any one of the datstructure,the best suited datastructure is "Queue".

Important methods associated with Queue

1. **boolean offer (Object obj)**

   => to add object into the Queues
2. **Object peek()**

   => It return the head element of the Queue

   If Queue is empty it returns null.
3. **Object element()**

   => It return the head element of the Queue

   If Queue is empty it throws NoSuchElementException.
4. **Object poll()**

   => It remove and return the head element of the queue

   If Queue is empty it returns null.
5. **Object remove()**

   => It remove and return the head element of the Queue

   If Queue is empty it returns NoSuchElementException.

PriorityQueue

1. To process the elements before processing, we need to store the elements based on some priority order
2. Priority Order

   => natural sorting order

   => customized sorting order
3. insertion order => not preseved based on some sorting it will be added.
4. duplicate => not allowed.
5. null insertion => not allowed

6. heterogenous => if we depend on natural sorting order,no objects should homogenous and it should implements Comparable if it is customized sorting order,then Object can be heterogenous and it need not implements Comparable.

Constructor

1. PriorityQueue p=new PriorityQueue();

   //Default Capacity=> 11

   //Insertion order => based on default natural sorting order.

2. PriorityQueue p=new PriorityQueue(int initialCapacity);
3. PriorityQueue p=new PriorityQueue(int initialCapacity,Comparator comparator)
4. PriorityQueue p=new PriorityQueue(SortedSet s);
5. PriorityQueue p=new PriorityQueue(Collection c);

eg#1.

```
import java.util.PriorityQueue;
public class TestApp{
    public static void main(String... args){
        PriorityQueue p=new PriorityQueue();
        System.out.println(p.poll());//null
        System.out.println(p.element());//NoSuchElementException
        for (int i=0;i<=10 ;i++ ){
            p.offer(i);
        }
        System.out.println(p);  //[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        System.out.println(p.poll());  //0
        System.out.println(p);  //[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    }
}
```

eg#2.

```
import java.util.PriorityQueue;
import java.util.Comparator;
public class TestApp{
    public static void main(String... args){
        PriorityQueue q=new PriorityQueue(15,new MyComparator());
        q.offer("Z");
        q.offer("A");
        q.offer("L");
        q.offer("B");
```

```
        System.out.println(q);//[Z,L,B,A]
    }
}
class MyComparator implements Comparator{
    @Override
    public int compare(Object obj1,Object obj2){
        String s1=obj1.toString();
        String s2=obj2.toString();
        return s2.compareTo(s1);
    }
}
```

Note: Some Operating System wont provide support for PriorityQueue(so the above program's output could vary).

1.6 V enchancement of Collection

1. **NavigableSet**

=> It is the child interface of SortedSet

=> It defines several methods for Navigation purposes

floor(e) => it returns the highest element which is <=e

lower(e) => it returns the highest element which is <e

ceiling(e) => it returns the lowest element which is >=e

higher (e) => it returns the lowest element which is <e

pollFirst() => remove and return first element

pollLast() => remove and return last element

descendingSet() =>returns NavigableSet in descending order.

eg#1.

```
import java.util.*;
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        TreeSet ts = new TreeSet();
        ts.add(1000);
        ts.add(2000);
        ts.add(3000);
```

```
        ts.add(4000);
        ts.add(5000);
        System.out.println(ts);//[1000, 2000, 3000, 4000, 5000]
        System.out.println("Ceiling value :: "+ts.ceiling(2000));//2000
        System.out.println("Higher value :: "+ts.higher(2000));//3000
        System.out.println("Floor value :: "+ts.floor(3000));//3000
        System.out.println("Lower value :: "+ts.lower(3000));//2000
        System.out.println("Poll First :: "+ts.pollFirst());//1000
        System.out.println(ts);//[2000,3000,4000,5000]
        System.out.println("Poll Last :: "+ts.pollLast());//5000
        System.out.println(ts);//[2000,3000,4000]
        System.out.println("DescendingSet :: "+ts.descendingSet());//
    }
}
```

2. **NavigableMap**

=> It defines several methods of Navigation purpose.

=> It is child interface of SortedMap.

NavigableMap defines the following methods

a. floorKey(e)
b. lowerKey(e)
c. ceilingKey(e)
d. higherKey(e)
e. pollFirstEntry()
f. pollLastEntry()
g. descendingMap()

eg#2.

```
import java.util.*;
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        TreeMap tm = new TreeMap();
        tm.put(10,"sachin");
        tm.put(7,"dhoni");
        tm.put(18,"kohli");
```

```
        tm.put(19,"dravid");
        tm.put(45,"rohith");
        //{7=dhoni, 10=sachin, 18=kohli, 19=dravid, 45=rohith}
        System.out.println(tm);
        System.out.println("Ceiling Key :: "+tm.ceilingKey(10));
        System.out.println("Higher Key :: "+tm.higherKey(10));
        System.out.println("Floor Key :: "+tm.floorKey(10));
        System.out.println("Lower Key :: "+tm.lowerKey(10));
        System.out.println("PollFirst :: "+tm.pollFirstEntry());
        System.out.println("PollLast :: "+tm.pollLastEntry());
        System.out.println("DescendingMap :: "+tm.descendingMap());
        System.out.println(tm);
    }
}
```

Output

{7=dhoni, 10=sachin, 18=kohli, 19=dravid, 45=rohith}

Ceiling Key :: 10

Higher Key :: 18

Floor Key :: 10

Lower Key :: 7

PollFirst :: 7=dhoni

PollLast :: 45=rohith

DescendingMap :: {19=dravid, 18=kohli, 10=sachin}

{10=sachin, 18=kohli, 19=dravid}

Collection vs Collections

Collections(c)

=> It is a utility class present in java.util package.
=> It defines the method meant for sorting,searching and reversing the elements

Note:

Collection(I)

|=> List

a. It won't speak about sorting,so use Collections(c).

|=> Set

a. If we want sorting then we can opt for TreeSet.

|=> Queue

a. If we want sorting then we can opt for PriorityQueue.

To sort the elemets of List

1. public static void sort(List l)**(default sorting if no 2nd argument is mentioned)**

   1. It sorts the element in ascending order/alphabetical order

   2.The elements should be homogenous and it should comparable otherwise it leads to ClassCastException.

   3.If it contains null,it would result in "NullPointerException".

2. public static void sort(List l,Comparator c)

   1. It sorts the elements based on our customization.

eg#1.

```
import java.util.*;

//Client Code

public class Test

{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add("Z");
        al.add("A");
        al.add("L");
        al.add("B");
        al.add("D");
        //al.add(new Integer(10)); :: ClassCastException
        //al.add(null); :: NullPointerException
        System.out.println("Before Sorting :: "+al);//[Z, A, L, B, D]
        //Collections
        Collections.sort(al);
        System.out.println("After Sorting :: "+al);//[A, B, D, L, Z]
    }

}
```

eg#2.

```
import java.util.*;
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add("Z");
        al.add("A");
        al.add("L");
        al.add("B");
        al.add("D");
        System.out.println("Before Sorting :: "+al);//[Z, A, L, B, D]
        //Collections
        Collections.sort(al,new MyComparator());
        System.out.println("After Sorting :: "+al);//[Z,L,D,B,A]
    }
}
class MyComparator implements Comparator
{
    @Override
    public int compare(Object obj1,Object obj2)
    {
        //logic for sorting
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        return -s1.compareTo(s2);
    }
}
```

binarySearch()

Searching Elements of List:

1. public static int binarySearch(List l, Object target);
   If we are Sorting List According to Natural Sorting Order then we have to Use this Method.
2. public static int binarySearch(List l, Object target, Comparator c);
   If we are Sorting List according to Comparator then we have to Use this Method.

Conclusions:

=> Internally the Above Search Methods **will Use Binary Search Algorithm.**

=> **Before performing Search Operation Compulsory List should be Sorted.** Otherwise we will get Unpredictable Results.

=> Successful Search Returns Index.

=> Unsuccessful Search Returns Insertion Point.

=> Insertion Point is the Location where we can Insert the Target Element in the SortedList.

=> If the List is Sorted according to Comparator then at the Time of Search Operation Also we should Pass the Same Comparator Object. Otherwise we will get Unpredictable Results.

```
import java.util.*;
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add("Z");
        al.add("A");
        al.add("M");
        al.add("K");
        al.add("a");
        //Collections
        Collections.sort(al);
        System.out.println("After Sorting :: "+al);//[A, K, M, Z, a]
        //BinarySearch :: success case -> index
        //BinarySearch :: failure case -> insertion point
        System.out.println("Index of Z is :: "+Collections.binarySearch(al,"Z"));
        System.out.println("Index of J is :: "+Collections.binarySearch(al,"J")); //-2
        System.out.println("Index of X is :: "+Collections.binarySearch(al,"X"));
    }
}
```

0 1. 2. 3. 4. → **success case**

[A, K, M, Z, a]

-1 -2. -3 -4. -5 → **failure case**

eg#2.

```
import java.util.*;
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(15);
        al.add(0);
        al.add(20);
        al.add(10);
        al.add(5);
        //Collections
        Collections.sort(al,new MyComparator());
        System.out.println("After Sorting :: "+al);//[20,15,10,5,0]
        //BinarySearch :: success case -> index
        //BinarySearch :: failure case -> insertion point
        System.out.println(Collections.binarySearch(al,10,new MyComparator()));//index
        System.out.println(Collections.binarySearch(al,13,new MyComparator()));//insertionpoint
        System.out.println(Collections.binarySearch(al,17,new MyComparator()));//insertionpoint
    }
}
class MyComparator implements Comparator
{
    @Override
    public int compare(Object obj1,Object obj2)
    {
        //logic for sorting
        Integer i1= (Integer)obj1;
        Integer i2= (Integer)obj2;
        return -i1.compareTo(i2);
    }
}
```

Output

After Sorting :: [20, 15, 10, 5, 0]

2

-3

-2

Eg: For the List of 3 Elements

A B Z

1. Range of Successful Search: 0 To 2
2. Range of Unsuccessful Search: **-4 To -1**
3. Total Result Range: **-4 To 2**

Note: For the List of n Elements

1. Successful Result Range: 0 To n-1
2. Unsuccessful Result Range: -(n+1) To -1
3. Total Result Range: -(n+1) To n-1

Reversing the Elements of List:

public static void reverse(List l);

reverse() Vs reverseOrder():

=> We can Use reverse() to Reverse Order of Elements of List.

=> We can Use reverseOrder() **to get Reversed Comparator.**

Comparator c1 = Collecctions.reverseOrder(Comparator c);

| |

Descending Order Ascending Order

eg#1.

```
import java.util.*;
public class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(15);
        al.add(0);
        al.add(20);
        al.add(10);
        al.add(5);
        Comparator c1 = new MyComparator();
```

```
        //Collections
        Collections.sort(al,c1);
        System.out.println("After Sorting :: "+al);//[20, 15, 10, 5, 0]
        Collections.reverse(al);
        System.out.println("After reversing :: "+al);//[0,5,10,15,20]
        Comparator c2 =Collections.reverseOrder(c1);
        Collections.sort(al,c2);
        System.out.println("ReverseOrder Sorting :: "+al);//[0,5,10,15,20]
    }
}
class MyComparator implements Comparator
{
    @Override
    public int compare(Object obj1,Object obj2)
    {
        //Sort:: Descending order
        Integer i1 = (Integer) obj1;
        Integer i2= (Integer) obj2;
        return -i1.compareTo(i2);
    }
}
```

Arrays

=> Arrays Class is an Utility Class to Define Several Utility Methods for Array Objects.

Sorting Elements of Array:

1. public static void sort(primitive[] p); To Sort According to Natural Sorting Order.
2. public static void sort(Object[] o); To Sort According to Natural Sorting Order.
3. public static void sort(Object[] o, Comparator c);**To Sort According to Customized Sorting Order.**

Note:

=> For Object Type Arrays we can Sort According to Natural Sorting Order OR Customized Sorting Order.

=> But we can Sort primitive[] Only Based on Natural Sorting.

eg#1.

```
import java.util.*;
public class Test
{
    public static void main(String[] args)
```

```
{
    int[] a= {10,5,20,11,6};
    System.out.println("Primitive Array before Sorting...");
    for (int data: a )
    {
        System.out.print(data+"\t");//10 5 20 11 6
    }
    System.out.println();

    //public static void sort(int[]);
    Arrays.sort(a);
    System.out.println("Primitive Array after Sorting...");
    for (int data: a )
    {
        System.out.print(data+"\t");//5 6 10 11 20
    }
    System.out.println("\n");

    String[] names = {"sachin","saurav","dhoni","kohli","azarudin"};
    System.out.println("Object Array before Sorting...");
    for (String data: names )
    {
        System.out.print(data+"\t");//sachin saurav dhoni kohli azarudin
    }
    System.out.println();

    //public static void sort(java.lang.Object[]);
    Arrays.sort(names);
    System.out.println("Object Array after Sorting...");
    for (String data: names )
    {
        System.out.print(data+"\t");//azarudin dhoni kohli sachin saurav
    }
    System.out.println();

    //public static void sort(T[], java.util.Comparator)
    Arrays.sort(names,new MyComparator());
    System.out.println("Object Array after Sorting using Comparator...");
    for (String data: names )
```

```
        {
            System.out.print(data+"\t");//saurav sachin kohli dhoni azarudin
        }
        System.out.println();
    }
}
class MyComparator implements Comparator
{
    @Override
    public int compare(Object obj1,Object obj2)
    {
        //Sort:: Descending order
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        return -s1.compareTo(s2);
    }
}
```

Output

Primitive Array before Sorting...

10 5 20 11 6

Primitive Array after Sorting...

5 6 10 11 20

Object Array before Sorting...

sachin saurav dhoni kohli azarudin

Object Array after Sorting...

azarudin dhoni kohli sachin saurav

Object Array after Sorting using Comparator...

saurav sachin kohli dhoni azarudin

Searching the Elements of Array

1. public static int binarySearch(primitive[] p, primitive target);

   If the Primitive Array Sorted According to Natural Sorting Order then we have to Use this Method.

2. public static int binarySearch(Object[] a, Object target);

   If the Object Array Sorted According to Natural Sorting Order then we have to Use this Method.

3. public static int binarySearch(Object[] a, Object target, **Comparator c**);

If the Object Array Sorted According to Comparator then we have to Use this Method.

Note: All Rules of Array Class binarySearch() are Exactly Same as Collections Class binarySearch().

eg#2.

```
import java.util.*;
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        int[] a= {10,5,20,11,6};
        //public static void sort(int[]);
        //5,6,10,11,20
        Arrays.sort(a);  //DNS
        //public static int binarySearch(int[], int);
        //success(key found) : return index
        //failure(key not found) : return insertion point
        System.out.println(Arrays.binarySearch(a,20));   //4
        System.out.println(Arrays.binarySearch(a,14));    //-5
        System.out.println();

        String names[]={"sachin","saurav","kohli","dhoni","azarudin"};
        //public static void sort(object[]);
        Arrays.sort(names);   //azarudin,dhoni,kohli,sachin,saurav
        //public static int binarySearch(Object[],Object);
        //success(key found) : return index
        //failure(key not found) : return insertion point
        System.out.println(Arrays.binarySearch(names,"kohli"));  //2
        System.out.println(Arrays.binarySearch(names,"raina"));  //-4
        System.out.println();

        //saurav sachin kohli dhoni azarudin
        Arrays.sort(names,new MyComparator());
        //public static int binarySearch(Object[],Object,Comparator c);
        //success(key found) : return index
        //failure(key not found) : return insertion point
        System.out.println(Arrays.binarySearch(names,"kohli",new MyComparator()));   //2
        System.out.println(Arrays.binarySearch(names,"raina",new MyComparator()));   //-3
    }
```

```
}
class MyComparator implements Comparator
{
    @Override
    public int compare(Object obj1,Object obj2)
    {
        //Sort:: Descending order
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        return -s1.compareTo(s2);
    }
}
```

Conversion of Array to List

Note: To convert the Collection to Array we have a method called Object[] toArray()

=> Arrays Class contains asList() for this

```
public static List asList(Object[] a);
```

=> Strictly Speaking this Method won't Create an Independent List Object, Just we are Viewing existing Array in List Form.

=> By using Array Reference if we Perform any Change Automatically that Change will be reflected to List Reference.

=> Similarly by using List Reference if we Perform any Change Automatically that Change will be reflected to Array.

=> By using List Reference if we are trying to Perform any Operation which Varies the Size then we will get Runtime Exception Saying UnsuportedOperationException.

eg#1.

```
import java.util.*;
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        //Array : size is fixed and it holds only homogenous type elements
        String arr[] = {"A","Z","B"};
        //Converting Array to List[Read only Array → can only set,we cannot add new as array has fixed size]
```

```
        List list = Arrays.asList(arr);
        System.out.println(list);  //[A,Z,B]
        System.out.println();

        System.out.println("Performing operation through Array");
        arr[0] = "K";
        System.out.println(list);  //[K,Z,B]
        System.out.println();

        System.out.println("Performing operation through List");
        list.set(1,"L");
        System.out.println(list);
        //Updating the element to List(indirectly to an Array)
        list.set(1,new Integer(10));
        System.out.println(list);   //ArrayStoreException(holds only homogenous type elements)
        //Adding the element to List(indirectly to an Array)
        list.add("sachin");
        System.out.println(list);   //UnSupportedOperationException(as array if of fixed size)
        //Remove the element from List(indirectly to an Array)
        list.remove(2);
        System.out.println(list);  //UnSupportedOperationException(as array if of fixed size)
    }
}
```

ConcurrentCollections

Objective

1. Majority of the Collection classes are not Threadsafety due to which there is a possiblity of "**DataIncosistency**" problem, so these collection classes are not ThreadSafe.
2. Already existing ThreadSafe Collection classes are
   a. Vector
   b. Hashtable
   c. synchronizedList(),synchronizedSet(),synchronizedMap() **of Collections class performance not upto the mark because it increases the waiting time for other threads even if they want to just do read operation.**

eg#1.

```
import java.util.ArrayList;
import java.util.Iterator;
public class Test {
```

```
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add("A");
        al.add("B");
        al.add("C");
        Iterator itr = al.iterator();
        while (itr.hasNext()){
            String s = (String)itr.next();
            System.out.println(s);
            al.add("D");        //java.util.ConcurrentModificationException
        }
    }
}
```

3. **In Normal Collections,Simultaneoulsy if one Thread is performing read operation ,other thread can't perform write operation if we try to do it would result in "ConccurentModificatinException" So normal Collections are not suitable for "MultiThreading Applications".**

eg#2.

```
import java.util.*;
//Client Code
public class Test extends Thread
{
    static ArrayList l=new ArrayList();
    @Override
    public void run()
    {
        //logic of a thread
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException ie)
        {
            ie.printStackTrace();
        }
        System.out.println("Child Thread updating list");
        l.add("B");
```

```
    }
    public static void main(String[] args) throws InterruptedException{
        l.add("A");
        l.add("B");
        l.add("C");
        Test t=new Test();
        t.start();
        //logic of main thread
        Iterator itr=l.iterator();
        while(itr.hasNext()){
            String data=(String)itr.next();
            System.out.println("Main Thread iterating list and the object is : "+data);
            Thread.sleep(3000);
        }
        System.out.println(l);
    }
}
```

Output

Main Thread iterating list and the object is : A

Child Thread updating list

Exception in thread "main" java.util.ConcurrentModificationException

at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)

at java.util.ArrayList$Itr.next(ArrayList.java:859)

at Test.main(Test.java:39)

Press any key to continue . . .

To resolve the above mentioned problems SUNMS introduced a new concpet in 1.5v called "ConcurrentCollections".

KeyPoints of ConcurrentCollections

1. Concurrent Collections are Always Thread Safe.
2. **When compared with Traditional Thread Safe Collections(like vector, hashtable, synchronizedList(),synchronizedSet(),synchronizedMap()) Performance is More because of different Locking Mechanism (segment locking/bucket locking mechanism).**
3. **While One Thread interacting Collection the Other Threads are allowed to Modify Collection in Safe Manner (best suited on Scalable Threading Application).**

Note: ConcurrentCollections -> Special package java.util.concurrent.*;

Most important ConcurrentCollections

1.ConcurrentHashMap
2.CopyonWriteArrayList
3.CopyonWriteArraySet

ConcurrentHashMap

Map(I)

| extends

ConcurrentMap(I)

| implements

ConcurrentHashMap(C)

Normal map => **put**(Object obj)

|=> If the key already present, then it will update the value for the corresponding key and old value will be returns.

ConcurrentMap => **putIfAbsent**(Object obj)

|=> If the key is already present,then it will not update the key with new value(otherwise inconsistencies will again occur).

eg#1. → put method of map will update the value if key already exist but ConcurrentHashMap provides a value which will update object only if the corresponding key doesn't exist.

```
import java.util.concurrent.*;
public class Test
{
    public static void main(String[] args){
        ConcurrentHashMap chm = new ConcurrentHashMap();
        chm.put(10,"sachin");
        chm.put(10,"afridi");
        System.out.println(chm);  //{10=afridi}
        chm.putIfAbsent(10,"sachin");
        System.out.println(chm);   //{10=afridi}
    }
}
```

Normal map => remove(**Object obj**)

|=> If the key already present, then it will remove that particular entry from the Map.

ConcurrentMap => remove(**Object Key,Object value**)

|=> **If both key and value matches only then that particular entry will be removed from the Map.**

eg#1.

```
import java.util.concurrent.*;
//Client Code
public class Test
{
    public static void main(String[] args){
        ConcurrentHashMap chm = new ConcurrentHashMap();
        chm.put(10,"sachin");
        chm.remove(10,"afridi");
        System.out.println(chm);    //{10=sachin}
        chm.remove(10,"sachin");
        System.out.println(chm);    //{}
    }
}
```

3. Normal map => replace(Object key,Object value)

|=> If the key already present, then update a new value.

ConcurrentMap => replace(Object Key,Object value,Object newValue)

|=> **If both key and value matches only then update the new value.**

eg#1.

```
import java.util.concurrent.*;
//Client Code
public class Test
{
    public static void main(String[] args){
        ConcurrentHashMap chm = new ConcurrentHashMap();
        chm.put(10,"sachin");
        chm.replace(10,"afridi","messi");
        System.out.println(chm);    //{10=sachin}
        chm.replace(10,"sachin","messi");
        System.out.println(chm);    //{10=messi}
    }
}
```

ConcurrentHashMap

=> Underlying Data Structure is Hashtable.

=> ConcurrentHashMap **allows Concurrent Read and Thread Safe Update Operations.**

=>**One of the key features of the ConcurrentHashMap is that it provides fine-grained locking, meaning that it locks only the portion of the map being modified, rather than the entire map. This makes it highly scalable and efficient for concurrent operations**

=> To Perform Read Operation Thread won't require any Lock. But to Perform Update Operation Thread requires Lock but it is the Lock of Only a Particular Part of Map (Bucket Level Lock).

=> Instead of Whole Map Concurrent Update achieved by Internally dividing Map into Smaller Portion which is defined by Concurrency.

=> The Default Concurrency Level is 16.

=> That is ConcurrentHashMap **Allows simultaneous Read Operation and simultaneously 16 Write (Update) Operations.**

=> null is Not Allowed for Both Keys and Values.

=> While One Thread iterating the Other Thread can Perform Update Operation and ConcurrentHashMap Never throw ConcurrentModificationException.

Constructors:

1. ConcurrentHashMap m = new ConcurrentHashMap();

   Creates an Empty ConcurrentHashMap with Default Initial Capacity 16 and Default Fill Ratio 0.75 and Default Concurrency Level 16.

2. ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity);
3. ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity, float fillRatio);
4. ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity, float fillRatio, int concurrencyLevel);
5. ConcurrentHashMap m = new ConcurrentHashMap(Map m);

eg#1.

```
import java.util.concurrent.ConcurrentHashMap;

class Test {
    public static void main(String[] args) {
        ConcurrentHashMap m = new ConcurrentHashMap();
        m.put(101, "A");
        m.put(102, "B");
        m.putIfAbsent(103, "C");
        m.putIfAbsent(101, "D");
        m.remove(101, "D");
        m.replace(102, "B", "E");
        System.out.println(m); //{103=C, 102=E, 101=A}
```

```
    }
}
```

eg#2.

```
import java.util.concurrent.*;
import java.util.*;
//Client Code
public class Test extends Thread
{
    static ConcurrentHashMap m = new ConcurrentHashMap(); // static because m is required in instance area(main, run) and static area.
    @Override
    public void run()
    {
        try {
            Thread.sleep(2000);
        }
        catch (InterruptedException e) {}
        System.out.println("Child Thread updating Map");
        m.put(103, "C");
    }
    public static void main(String[] args) throws Exception
    {
        m.put(101, "A");
        m.put(102, "B");
        Test t =new Test();
        t.start();
        Set s = m.keySet();
        Iterator itr = s.iterator();
        while (itr.hasNext())
        {
            Integer I1 = (Integer) itr.next();
            System.out.println("Main Thread iterating and Current Entry is:"+I1+"....." + m.get(I1));
            Thread.sleep(3000);
        }
        System.out.println(m);
    }
```

}

Output

Main Thread iterating and Current Entry is:101.....A

Child Thread updating Map

Main Thread iterating and Current Entry is:102.....B

Main Thread iterating and Current Entry is:103.....C

{101=A, 102=B, 103=C}

=> Update and we won't get any ConcurrentModificationException.

=> If we Replace ConcurrentHashMap with HashMap then we will get ConcurrentModificationException.

Difference b/w HashMap and ConcurrentHashMap

HashMap => Not Thread Safe

ConcurrentHashMap => Thread Safe

HashMap => Performance is high as Threads are not required to wait to operate.

ConcurrentHashMap => Performance is low as Threads are required to operate.

HashMap => One Thread while Operating on HashMap,other thread are not allowed to modify if it tries to do it would result in "ConcurrentModificationException".

ConcurrentHashMap => One Thread while Opeating on HashMap,other thread are allowed to modify in safe manner, it wont throw "ConcurrentModificationException".

HashMap => Iterator of HashMap if **FailFast(meaning it will lead to ConcurrentModificationException and iterator will fail).**

ConcurrentHashMap => Iterator of ConcurrentHashMap is **FailSafe.(meaning it will not lead to ConcurrentModificationException and iterator will not fail).**

HashMap => null is allowed for both keys and values.

ConcurrentHashMap => null is not allowed for both keys and values.

HashMap => Introduced in **1.2v**

ConcurrentHashMap => Introduced in **1.5v**

Difference b/w ConcurrentHashMap,synchronizedMap() and Hashtable

ConcurrentHashMap =>Thread safety without putting lock on entire Object **lock at bucket level**

synchronizedMap() =>Thread safety by putting lock on entire Object.

Hashtable =>Thread safety by putting lock on entire Object.

ConcurrentHashMap =>Mulitple Threads are allowed to operate on Object in same manner

synchronizedMap() =>Only One Thread is allowed to operate on Object

Hashtable =>Only One Thread is allowed to operate on Object

ConcurrentHashMap =>Read operation can be performed without any Lock,to perform update/write we need Object level lock.

synchronizedMap() =>To perform read and update we need Object level lock

Hashtable =>To perform read and update we need Object level lock

ConcurrentHashMap =>No ConcurrentModificationException

synchronizedMap() =>It would result in ConcurrentModificationException

Hashtable =>It would result in ConcurrentModificationException

ConcurrentHashMap =>Iterator is **FailSafe**

synchronizedMap() =>Iterator is **FailFast**

Hashtable =>Iterator is **FailFast**

ConcurrentHashMap =>null not allowed for both keys and values

synchronizedMap() =>null allowed for keys and values

Hashtable =>null not allowed for both keys and values

ConcurrentHashMap =>JDK1.5V

synchronizedMap() =>JDK1.2V

Hashtable =>JDK1.0V

CopyOnWriteArrayList

Collection(I)

| extends

List(I)

| implements

CopyOnWriteArrayList(c)

package :: java.util.concurrent.CopyOnWriteArrayList(c)

=> It is a Thread Safe Version of ArrayList as the Name indicates **CopyOnWriteArrayList Creates a Cloned Copy of Underlying ArrayList for Every Update Operation at Certain Point Both will Synchronized Automatically Which is taken Care by JVM Internally.**

=> As Update Operation will be performed on cloned Copy there is No Effect for the Threads which performs Read Operation.

=> It is Costly to Use because for every Update Operation a cloned Copy will be Created. Hence CopyOnWriteArrayList is the Best Choice if Several Read Operations and Less Number of Write Operations are required to Perform.

=> Insertion Order is Preserved.

=> Duplicate Objects are allowed.

=> Heterogeneous Objects are allowed.

=> null Insertion is Possible.

=> It implements Serializable, Clonable and RandomAccess Interfaces.

=> While One Thread iterating CopyOnWriteArrayList, the Other Threads are allowed to Modify and we won't get ConcurrentModificationException. That is iterator is FailSafe.

=> Iterator of ArrayList can Perform Remove Operation but Iterator of CopyOnWriteArrayList can't Perform Remove Operation. Otherwise we will get RuntimeException Saying UnsupportedOperationException.

Constructors:

1. CopyOnWriteArrayList l = new CopyOnWriteArrayList();
2. CopyOnWriteArrayList l = new CopyOnWriteArrayList(Collection c);
3. CopyOnWriteArrayList l = new CopyOnWriteArrayList(Object[] a);

Methods

boolean addIfAbsent(Object o): The Element will be Added if and Only if List doesn't contain this Element.

```
CopyOnWriteArrayList l = new CopyOnWriteArrayList();

l.add("A");

l.add("A");

l.addIfAbsent("B");

l.addIfAbsent("B");

System.out.println(l); //[A, A, B]
```

int addAllAbsent(Collection c): The Elements of Collection will be Added to the List if Elements are Absent and Returns Number of Elements Added.

```
ArrayList l = new ArrayList();
l.add("A");
l.add("B");
CopyOnWriteArrayList l1 = new CopyOnWriteArrayList();
l1.add("A");
l1.add("C");
System.out.println(l1); //[A, C]
l1.addAll(l);  //Adding ArrayList 'l'
System.out.println(l1); //[A, C, A, B]
ArrayList l2 = new ArrayList();
l2.add("A");
l2.add("D");
l1.addAllAbsent(l2);
System.out.println(l1); //[A, C, A, B, D]
```

eg#1.

```
import java.util.concurrent.*;
import java.util.*;
//Client Code
public class Test extends Thread
{
    static CopyOnWriteArrayList cowal = new CopyOnWriteArrayList();
    @Override
    public void run()
    {
        try {
            Thread.sleep(2000);
        }
        catch (InterruptedException e) {}
        System.out.println("Child Thread updating List");
        cowal.add("C");
    }
    public static void main(String[] args)throws Exception
    {
        cowal.add("A");
        cowal.add("B");
```

```
        Test t = new Test();
        t.start();
        Iterator itr = cowal.iterator();
        while (itr.hasNext()) {
            String l1 = (String) itr.next();
            System.out.println ("Main Thread iterating and Current Object is :: "+l1);
            Thread.sleep(3000);
        }
        System.out.println(cowal);
    }
}
```

Ouput

Main Thread iterating and Current Object is :: A

Child Thread updating List

Main Thread iterating and Current Object is :: B

[A, B, C]

=> In the Above Example while Main Thread iterating List Child Thread is allowed to Modify and we won't get any ConcurrentModificationException.

=> If we Replace CopyOnWriteArrayList with ArrayList then we will get ConcurrentModificationException.

eg#2.

```
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.Iterator;
class Test {
    public static void main(String[] args){
        CopyOnWriteArrayList l = new CopyOnWriteArrayList();
        l.add("A");
        l.add("B");
        l.add("C");
        l.add("D");
        System.out.println(l); //[A, B, C, D]
        Iterator itr = l.iterator();
        while (itr.hasNext()) {
            String s = (String)itr.next();
            if (s.equals("D"))
                itr.remove();   //RE: java.lang.UnsupportedOperationException
```

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

Iterator of CopyOnWriteArrayList can't Perform Remove Operation. Otherwise we will get RuntimeException: UnsupportedOperationException.

eg#3.

```
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.Iterator;
class Test {
    public static void main(String[] args) {
        CopyOnWriteArrayList l = new CopyOnWriteArrayList();
        l.add("A");
        l.add("B");
        l.add("C");
        Iterator itr = l.iterator();
        l.add("D");
        while (itr.hasNext()) {
            String s = (String)itr.next();
            System.out.println(s); //[A,B,C]
        }
    }
}
```

Arraylist vs CopyOnWriteArrayList

ArrayList => It is Not Thread Safe.

CopyOnWriteArrayList=> It is Thread Safe because Every Update Operation will be performed on Separate cloned Copy.

ArrayList => While One Thread iterating List Object, the Other Threads are Not allowed to Modify List Otherwise we will get ConcurrentModificationException.

CopyOnWriteArrayList => While One Thread iterating List Object, the Other Threads are allowed to Modify List in Safe Manner and we won't get ConcurrentModificationException.

ArrayList => Iterator is Fail-Fast.

CopyOnWriteArrayList => Iterator is Fail-Safe.

ArrayList => Iterator of ArrayList can Perform Remove Operation.

CopyOnWriteArrayList => Iterator of CopyOnWriteArrayList can't Perform Remove Operation Otherwise we will get RuntimeException: UnsupportedOperationException.

ArrayList => Introduced in 1.2 Version.

CopyOnWriteArrayList => Introduced in 1.5 Version.

Difference b/w CopyOnWriteArrayList,synchornizedList and Vector

CopyOnWriteArrayList

We will get Thread Safety because Every Update Operation will be performed on Separate cloned Copy.

At a Time Multiple Threads are allowed to Access/ Operate on CopyOnWriteArrayList.

While One Thread iterating List Object, the Other Threads are allowed to Modify Map and we won't get ConcurrentModificationException.

Iterator is Fail-Safe and won't raise ConcurrentModificationException.

Iterator can't Perform Remove Operation Otherwise we will get UnsupportedOperationException.

Introduced in 1.5 Version.

synchronizedList

We will get Thread Safety because at a Time List can be accessed by Only One Thread at a Time.

At a Time Only One Thread is allowed to Perform any Operation on List Object.

While One Thread iterating , the Other Threads are Not allowed to Modify List. Otherwise we will get ConcurrentModificationException.

Iterator is Fail-Fast and it will raise ConcurrentModificationException.

Iterator canPerform Remove Operation.

Introduced in 1.2 Version.

Vector

We will get Thread Safety because at a Time Only One Thread is allowed to Access Vector Object.

At a Time Only One Thread is allowed to Operate on Vector Object.

While One Thread iterating, the Other Threads are Not allowed to Modify Vector. Otherwise we will get ConcurrentModificationException.

Iterator is Fail-Fast and it will raise ConcurrentModificationException.

Iterator can Perform Remove Operation.

Introduced in 1.0 Version

CopyOnWriteArraySet

Collection (I)

|

Set (I)

|

CopyOnWriteArraySet (C)

=> It is a Thread Safe Version of Set.

=> Internally Implement by CopyOnWriteArrayList.

=> Insertion Order is Preserved.

=> Duplicate Objects are Not allowed.

=> Multiple Threads can Able to Perform Read Operation simultaneously but for Every Update Operation a Separate cloned Copy will be Created.

=> As for Every Update Operation a Separate cloned Copy will be Created which is Costly Hence if Multiple Update Operation are required then it is Not recommended to Use CopyOnWriteArraySet.

=> While One Thread iterating Set the Other Threads are allowed to Modify Set and we won't get ConcurrentModificationException.

=> Iterator of CopyOnWriteArraySet can Perform Only Read Operation and won't Perform Remove Operation. Otherwise we will get RuntimeException: UnsupportedOperatonException.

Constructors:

1. CopyOnWriteArraySets = new CopyOnWriteArraySet();
   Creates an Empty CopyOnWriteArraySet Object.
2. CopyOnWriteArraySet s = new CopyOnWriteArraySet(Collection c);
   Creates CopyOnWriteArraySet Object which is Equivalent to given Collection Object.

Methods:Whatever Methods Present in Collection and Set Interfaces are the Only Methods Applicable for CopyOnWriteArraySet and there are No Special Methods.

```
import java.util.concurrent.CopyOnWriteArraySet;

class Test {
    public static void main(String[] args) {
        CopyOnWriteArraySet s = new CopyOnWriteArraySet();
        s.add("A");
        s.add("B");
        s.add("C");
        s.add("A");
        s.add(null);
        s.add(10);
        s.add("D");
        System.out.println(s);//[A, B, C, null, 10, D]
    }
}
```

Differences between CopyOnWriteArraySet() and synchronizedSet()
CopyOnWriteArraySet()

It is Thread Safe because Every Update Operation will be performed on Separate Cloned Copy.
While One Thread iterating Set, the Other Threads are allowed to Modify and we won't get ConcurrentModificationException.
Iterator is Fail Safe.
Iterator can Perform Only Read Operation and can't Perform Remove Operation Otherwise we will get RuntimeException Saying UnsupportedOperationException.
Introduced in 1.5 Version.

synchronizedSet()

It is Thread Safe because at a Time Only One Thread can Perform Operation.

While One Thread iterating, the Other Threads are Not allowed to Modify Set Otherwise we will get ConcurrentModificationException.

Iterator is Fail Fast.

Iterator can Perform Both Read and Remove Operations.

Introduced in 1.7 Version.

Difference FailFast vs FailSafe Iterators

Fail Fast Iterator: While One Thread iterating Collection if Other Thread trying to Perform any Structural Modification to the underlying Collection then immediately Iterator Fails by raising ConcurrentModificationExcepion. Such Type of Iterators are Called Fail Fast Iterators.

```
import java.util.ArrayList;

import java.util.Iterator;

class Test {
    public static void main(String[] args) {
        ArrayList l = new ArrayList();
        l.add("A");
        l.add("B");
        Iterator itr = l.iterator();//FailFast Iterator
        while(itr.hasNext()) {
            String s = (String)itr.next();
            System.out.println(s); //A
            l.add("C"); // java.util.ConcurrentModificationException
        }
    }
}
```

Note: Internally Fail Fast Iterator will Use Some Flag named with **MOD** **to Check underlying Collection is Modified OR Not while iterating.**

Fail Safe Iterator:

=> While One Thread iterating if the Other Threads are allowed to Perform any Structural Changes to the underlying Collection, Such Type of Iterators are Called Fail Safe Iterators.

=> Fail Safe Iterators won't raise ConcurrentModificationException because Every Update Operation will be performed on Separate cloned Copy.

```
import java.util.concurrent.CopyOnWriteArraySet;

import java.util.Iterator;

class Test {
    public static void main(String[] args) {
        CopyOnWriteArraySet l = new CopyOnWriteArraySet();
        l.add("A");
        l.add("B");
        Iterator itr = l.iterator();//FailSafe Iterator
        while(itr.hasNext()) {
            String s = (String)itr.next();
            System.out.println(s); //A B
            l.add("C");
        }
        System.out.println(l);//[A,B,C]
    }
}
```

Interfaces

1. Collection
2. List -> ArrayList,LinkedList :: add(int, object), set(int,object), remove(int,object)
   -> LegacyClasses :: Stack,Vector
   -> CopyOnWriteArrayList[Avoids ConcurrentModification]
3. Set -> HashSet,LinkedHashSet :: add(object),set(object),remove(object)
   -> CopyOnWriteArraySet[Avoids ConcurrentModification]
4. NavigableSet -> TreeSet
5. SortedSet -> TreeSet
6. Map(K,V) → HashMap,LinkedHashMap,WeakHashMap,IdentityHashMap,Hastable,Properties :: put(object,object),get(Object)
   -> ConcurrentHashMap[Avoids ConcurrentModification]
7. NavigableMap -> TreeMap
8. SortedMap -> TreeMap

Interfaces related to Sorting

1. Comparable meant for DefaultNatural Sorting order of Objects int compareTo(Object obj)

2. Comparator meant for Custom Sorting int compare(Object obj1,Object obj2)

Cursors related to iteration

1. Enumeration :: iterators for legacy classes[read operation only]
2. Iterator :: Universal cursor[read in forward direction and removal operation] :: hasNext(),next()
3. ListIterator :: iterators for List objects[read(forward,backward),read,update,remove]

UtilityClasses

1.Collections :: sort(),binarySearch(),reverse(),reverseOrder()

2.Arrays :: sort(),binarySearch(),asList()

LambdaExpression in Collection

1. **List(I)**
   -> ArrayList,LinkedList,Vector,Stack
   -> Insertion order is preserved.
   -> Duplicate Objects are allowed.
2. **Set(I)**
   -> HashSet,TreeSet(based on some sorting)
   -> Insertion order is not preserved becoz data would be stored based on HashCode.
   -> Duplicate Objects are not allowed.
3. **Map(I)**
   -> HashMap,TreeMap(based on some sorting)
   -> Data would be stored in the form of K,V pair
   -> Insertion order is not preserved becoz data would be stored based on HashCode of Key.
   -> Duplicate Objects are not allowed.

Comparator(I)

-> It contains only one abstract method called "compare()".

-> To define our own sorting we need to use Customized Sorting.

-> public int compare(Object obj1,Object obj2)

=> return -ve iff obj1 has to come before obj2.

=> return +ve iff obj1 has to come after obj2.

=> return 0 iff obj1 and obj2 are equal.

SortedCollections

1. Sorted List
2. Sorted Map
3. Sorted Set

Sorted List

List(may be ArrayList,LinkedList,Vector or Stack) never talks about sorting order.

If we want sorting for the list then we should use Collections class sort() method.

Collections.sort(list)==>meant for Default Natural Sorting Order

For String objects: Alphabetical Order

For Numbers : Ascending order

Demo Program to Sort elements of ArrayList according to Defaut Natural Sorting Order

(Ascending Order):

```
import java.util.ArrayList;
import java.util.Collections;
class Test
{
    public static void main(String[] args)
    {
        ArrayList l = new ArrayList();
        l.add(10);
        l.add(0);
        l.add(15);
        l.add(5);
        l.add(20);
        System.out.println("Before Sorting:"+l);
        Collections.sort(l);
        System.out.println("After Sorting:"+l);
    }
}
```

Output:

Before Sorting:[10, 0, 15, 5, 20]

After Sorting:[0, 5, 10, 15, 20]

Instead of Default natural sorting order if we want customized sorting order then we should go for Comparator interface.

Comparator interface contains only one abstract method: compare(). Hence it is Functional interface.

public int compare(obj1,obj2)

returns -ve iff obj1 has to come before obj2

returns +ve iff obj1 has to come after obj2

returns 0 iff obj1 and obj2 are equal

Collections.sort(list,Comparator)==>meant for Customized Sorting Order

Demo Program to Sort elements of ArrayList according to Customized Sorting Order

(Descending Order):

```
import java.util.TreeSet;
import java.util.Comparator;
class MyComparator implements Comparator
{
    public int compare(Integer I1,Integer I2)
    {
        if(I1<I2)
        {
            return +1;
        }
        else if(I1>I2)
        {
            return -1;
        }
        else
        {
            return 0;
        }
    }
}
class Test
{
    public static void main(String[] args)
    {
        TreeSet I = new TreeSet(new MyComparator());
        I.add(10);
        I.add(0);
        I.add(15);
        I.add(5);
        I.add(20);
        System.out.println(I);
    }
}
```

//Descending order Comparator

Output: [20, 15, 10, 5, 0]

Shortcut way: Using generics

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.Collections;
class MyComparator implements Comparator
{
    public int compare(Integer I1,Integer I2)
    {
        return (I1>I2)?-1:(I1<I2)?1:0;
    }
}
class Test
{
    public static void main(String[] args)
    {
        ArrayList<Integer> l = new ArrayList<Integer>();
        l.add(10);
        l.add(0);
        l.add(15);
        l.add(5);
        l.add(20);
        System.out.println("Before Sorting:"+l);
        Collections.sort(l,new MyComparator());
        System.out.println("After Sorting:"+l);
    }
}
```

Sorting with Lambda Expressions + Generics

As Comparator is Functional interface, we can replace its implementation with Lambda Expression

Collections.sort(l,(I1,I2)->(I1<I2)?1:(I1>I2)?-1:0);

Demo Program to Sort elements of ArrayList according to Customized Sorting Order By using Lambda Expressions(Descending Order):

```
import java.util.ArrayList;
import java.util.Collections;
class Test
```

```
{
    public static void main(String[] args)
    {
        ArrayList l= new ArrayList();
        l.add(10);
        l.add(0);
        l.add(15);
        l.add(5);
        l.add(20);
        System.out.println("Before Sorting:"+l);
        Collections.sort(l,(l1,l2)->(l1<l2)?1:(l1>l2)?-1:0);
        System.out.println("After Sorting:"+l);
    }
}
```

Output:

Before Sorting:[10, 0, 15, 5, 20]

After Sorting:[20, 15, 10, 5, 0]

Shortcut code

```
import java.util.*;
public class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add(0);
        al.add(5);
        al.add(15);
        al.add(20);
        System.out.println("Before Sorting :: "+al);
        //Functional Programming :: passing function body as argument
        Collections.sort(al,(i1,i2)->-i1.compareTo(i2));
        System.out.println("After Sorting :: "+al);
    }
}
```

Output

Before Sorting :: [10, 0, 5, 15, 20]

After Sorting :: [20, 15, 10, 5, 0]

After using generics the warning will also go away which comes after compilation

Sorting for Customized class objects by using Lambda Expressions

without lambda Expression

```
import java.util.*;
class Employee
{
    int eno;
    String ename;
    Employee(int eno,String ename){
        this.eno=eno;
        this.ename=ename;
    }
    public String toString(){
        return eno+"==>"+ename;
    }
}
class EmployeeComparator implements Comparator
{
    public int compare(Object obj1,Object obj2){
        Employee e1=(Employee) obj1;
        Employee e2=(Employee) obj2;
        return e1.eno>e2.eno? -1 : e1.eno<e2.eno ? +1:0;
    }
}
public class Test {
    public static void main(String[] args){
        ArrayList al=new ArrayList();
        al.add(new Employee(10,"sachin"));
        al.add(new Employee(14,"ponting"));
        al.add(new Employee(7,"dhoni"));
        al.add(new Employee(9,"lara"));
        al.add(new Employee(18,"kohli"));
        System.out.println("Before sorting :"+al);
```

```
        Collections.sort(al,new EmployeeComparator());
        System.out.println("After sorting :"+al);
    }


}
Before sorting :[10>sachin, 14>ponting, 7>dhoni, 9>lara, 18>kohli]
After sorting :[18>kohli, 14>ponting, 10>sachin, 9>lara, 7>dhoni]
```

With Lambda Expression

```
import java.util.*;
class Employee
{
    int eno;
    String ename;
    Employee(int eno,String ename){
        this.eno=eno;
        this.ename=ename;
    }
    public String toString(){
        return eno+"==>"+ename;
    }
}
public class Test {
    public static void main(String[] args){
        ArrayList al=new ArrayList();
        al.add(new Employee(10,"sachin"));
        al.add(new Employee(14,"ponting"));
        al.add(new Employee(7,"dhoni"));
        al.add(new Employee(9,"lara"));
        al.add(new Employee(18,"kohli"));
        System.out.println("Before sorting :"+al);
        Collections.sort(al,(e1,e2)-> e1.eno>e2.eno?-1:e1.eno<e2.eno?+1:0);
        System.out.println("After sorting :"+al);
    }
}
Before sorting :[10>sachin, 14>ponting, 7>dhoni, 9>lara, 18==>kohli]
After sorting :[18>kohli, 14>ponting, 10>sachin, 9>lara, 7==>dhoni]
```

Shortcut code

```
class Employee
{
    Integer eno;
    String ename;
    Employee(Integer eno,String ename)
    {
        this.eno = eno;
        this.ename =ename;
    }
    public String toString()
    {
        return eno + "---" + ename;
    }
}
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(new Employee(10,"sachin"));
        al.add(new Employee(7,"dhoni"));
        al.add(new Employee(18,"kohli"));
        al.add(new Employee(19,"dravid"));
        al.add(new Employee(45,"rohith"));
        System.out.println("Before Sorting :: "+al);
        //Functional Programming :: passing function body as argument
        Collections.sort(al,
            (e1,e2) -> -e1.eno.compareTo(e2.eno)
        );
        System.out.println("After Sorting :: "+al);
    }
}
```

output

Before Sorting :: [10---sachin, 7---dhoni, 18---kohli, 19---dravid, 45---rohith]

After Sorting :: [45---rohith, 19---dravid, 18---kohli, 10---sachin, 7---dhoni]

2. **Sorted Set**

In the case of Set, if we want Sorting order then we should go for TreeSet

1. TreeSet t = new TreeSet();
   This TreeSet object meant for default natural sorting order
2. TreeSet t = new TreeSet(Comparator c);
   This TreeSet object meant for Customized Sorting Order

Demo Program for Default Natural Sorting Order(Ascending Order):

```
import java.util.TreeSet;
class Test
{
public static void main(String[] args)
{
    TreeSet t = new TreeSet();
    t.add(10);
    t.add(0);
    t.add(15);
    t.add(5);
    t.add(20);
    System.out.println(t);
    }
}
```

Output: [0, 5, 10, 15, 20]

Demo Program for Customized Sorting Order(Descending Order):

```
import java.util.TreeSet;
class Test
{
    public static void main(String[] args)
    {
        TreeSet t = new TreeSet((l1,l2) → (l1>l2) ? -1 : (l1 ?  1 : 0);
        t.add(10);
        t.add(0);
        t.add(15);
        t.add(25);
        t.add(5);
```

```
        t.add(20);
        System.out.println(t);
    }
}
```

Output: [25, 20, 15, 10, 5, 0]

3. **Sorted Map:**

In the case of Map, if we want default natural sorting order of keys then we should go for TreeMap.

1. TreeMap m = new TreeMap();
   This TreeMap object meant for default natural sorting order of keys
2. TreeMap t = new TreeMap(Comparator c);
   This TreeMap object meant for Customized Sorting Order of keys

Demo Program for Default Natural Sorting Order(Ascending Order):

```
import java.util.*;
public class Test {
    public static void main(String[] args){
        TreeMap<Integer,String> t =new TreeMap<Integer,String>();
        t.put(10,"sachin");
        t.put(14,"ponting");
        t.put(18,"kohli");
        t.put(9,"lara");
        t.put(17,"ABD");
        t.put(7,"dhoni");
        System.out.println(t);
    }
}
```

{7=dhoni, 9=lara, 10=sachin, 14=ponting, 17=ABD, 18=kohli}

Demo Program for Customized Sorting Order(Descending Order):

```
import java.util.*;
public class Test {
    public static void main(String[] args){
        TreeMap<Integer,String> t =new TreeMap<Integer,String>((I1,I2) -> I1>I2 ? -1 : I1<I2 ? +1 : 0);
        t.put(10,"sachin");
        t.put(14,"ponting");
        t.put(18,"kohli");
```

```
        t.put(9,"lara");
        t.put(17,"ABD");
        t.put(7,"dhoni");
        System.out.println("After sorting ::"+t);
    }
}
```

After sorting ::{18=kohli, 17=ABD, 14=ponting, 10=sachin, 9=lara, 7=dhoni}

Stream API

diagarm (11feb)++++++++(not added)

package name : java.util.stream.*;

Streams

To process objects of the collection**(like finding only the types of laptops whose price < 3000 from an arrayList and then printing those pcs.)**, in 1.8 version Streams concept introduced.

What is the differences between java.util.streams and java.io streams?

=> java.util streams meant for processing objects from the collection. Ie, it **represents a stream of objects** from the collection.

=> Java.io streams meant for processing binary and character data with respect to file. i.e it represents stream of binary data or character data from the file.**(imp diagram missing - 11feb)**

=> Hence java.io streams and java.util streams both are different.

What is the difference between collection and stream?

=> If we want to represent a group of individual objects as a single entity then We should go for collection.

=> If we want to process a group of objects from the collection then we should go for streams.

=> We can create a stream object to the collection by using stream() method of Collection interface. **stream() method is a default method** added to the Collection in 1.8 version.

=> **Stream is an interface present in java.util.stream**. Once we got the stream, by using that we can process objects of that collection.

We can process the objects in the following 2 phases

1.Configuration

2.Processing

1. **Configuration:**

We can configure either by using filter mechanism or by using map mechanism.

Filtering: We can configure a filter to filter elements from the collection based on some boolean condition by using filter()method of Stream interface.

public Stream filter(Predicate t)

here (Predicate t) can be a boolean valued function/lambda expression

|-> boolean test(T t)

Mapping:

If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for map() method of Stream interface.

public Stream map (Function f); **// It can be lambda expression**

also

|-> R apply(T t)

Ex:

Stream s = c.stream();

Stream s1 = s.map(i-> i+10);

Once we performed configuration we can process objects by using several methods.

eg#1.

```
import java.util.*;
import java.util.stream.*;
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(0);
        al.add(5);
        al.add(10);
        al.add(15);
        al.add(20);
        al.add(25);
        ArrayList doubleList = new ArrayList();
        //Normal way
        for ( Integer i1: al )
            doubleList.add(i1*2);
        System.out.println(doubleList);

        System.out.println("Using Stream API*");
        List<Integer> stremCode = al.stream()    //creating a stream
```

```
                                .map(i->i%2==0).  //Configuration : new object
                                .collect(Collectors.toList());     //Creating a list
        System.out.println(streamCode);
    }
}
```

Output

[0, 10, 20, 30, 40, 50]

Using Stream API*

[0, 10, 20, 30, 40, 50]

eg#2.

```
import java.util.*;
import java.util.stream.*;
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        List al = new ArrayList();
        al.add(10);
        al.add(15);
        al.add(20);
        al.add(0);
        al.add(100);
        al.add(25);
        int result=sumIterator(al);
        System.out.println("The sum is :: " + result);
        System.out.println("Using Stream API");
        System.out.println("The sum is :: " + sumIteratorUsingStream(al));
    }
    public static int sumIterator(List al)
    {
        //Without using Stream API
        int sum = 0;
        Iterator itr = al.iterator();
        while (itr.hasNext())
        {
```

```
            int num = itr.next();
            if (num>10)
            {
                sum+=num;
            }
        }
        return sum;
    }
    public static int sumIteratorUsingStream(List al)
    {
        //Using Stream API
        return al.stream().filter(i->i>10).mapToInt(i->i).sum();
    }
}
```

- **Without `mapToInt`:** If we directly use `sum()` after the filter, it would attempt to sum the filtered elements as objects (assuming `al` holds objects). This might lead to unexpected behavior depending on the object's implementation.
- **With `mapToInt`:**
  - The `mapToInt(i->i)` part essentially converts each filtered element (`i`) into its integer value. This ensures the `sum()` method operates on integers, providing the correct sum of the filtered elements.

eg#2.

```
import java.util.*;
import java.util.stream.*;
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        List al = new ArrayList();
        al.add("sachin");
        al.add("saurav");
        al.add("dhoni");
        al.add("kohli");
        al.add("yuvi");
        printObjectData(al);
        printObjectDataUsingStream(al);
    }
```

```
    public static void printObjectData(List al)
    {
        List result = new ArrayList();
        //Without using Stream API
        for (String name :al )
        {
            result.add(name.toUpperCase());
        }
        System.out.println(result);
    }
    public static void printObjectDataUsingStream(List al)
    {
            //Using Stream API
            List result =
                    al.stream()
                    .map(name->name.toUpperCase())
                    .collect(Collectors.toList());
            System.out.println(result);
    }
}
```

Output

[SACHIN, SAURAV, DHONI, KOHLI, YUVI]

[SACHIN, SAURAV, DHONI, KOHLI, YUVI]

2. **Processing → +diagram left(configuration + processing one)**

processing by collect() method

```
import java.util.*;
import java.util.stream.*;
class Employee
{
    int eid;
    String ename;
    float esal;
    Employee(int eid,String ename,float esal)
    {
        this.eid = eid;
        this.ename = ename;
```

```
        this.esal = esal;
    }
    public String toString()
    {
        return eid+"->"+ename+"->" +esal;
    }
}
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        List al = new ArrayList();
        al.add(new Employee(10,"sachin",35000f));
        al.add(new Employee(7,"dhoni",30000f));
        al.add(new Employee(18,"kohli",35000f));
        al.add(new Employee(1,"rahul",28000f));
        al.add(new Employee(19,"dravid",30000f));
        al.add(new Employee(45,"rohith",25000f));
        System.out.println(al);
        printObjectDataUsingStream(al);
    }
    public static void printObjectDataUsingStream(List al)
    {
        //Using Stream API
        List result =
                    al.stream()
                    .filter(emp->emp.esal<35000f)
                    .map(emp->emp.esal)
                    .collect(Collectors.toList());
        System.out.println(result);
        System.out.println();
        Set noDuplicates =
                    al.stream()
                    .filter(emp->emp.esal<35000f)
                    .map(emp->emp.esal)
                    .collect(Collectors.toSet()); //to remove duplicates
        System.out.println(noDuplicates);
```

```
        System.out.println();
        Map<Integer,String> map =
                al.stream()
                .filter(emp->emp.esal<35000f). //no map here as we want key-value pair
                .collect(Collectors.toMap(emp->emp.eid,emp->emp.ename));
        System.out.println(map);
    }
}
```

Output

[10->sachin->35000.0, 7->dhoni->30000.0, 18->kohli->35000.0, 1->rahul->28000.0, 19-

dravid->30000.0, 45->rohith->25000.0]
[30000.0, 28000.0, 30000.0, 25000.0]
[25000.0, 30000.0, 28000.0]
{1=rahul, 19=dravid, 7=dhoni, 45=rohith}

b.Processing by count()method

This method returns number of elements present in the stream.

```
public long count()
```

c.Processing by sorted()method

If we sort the elements present inside stream then we should go for sorted() method. The sorting can either default natural sorting order or customized sorting order specified by comparator.

sorted()- default natural sorting order

sorted(Comparator c)-customized sorting order.

d.Processing by min() and max() methods

min(Comparator c)

returns minimum value according to specified comparator.

max(Comparator c)

returns maximum value according to specified comparator.

e.forEach() method

This method will not return anything. This method will take lambda expression as argument and apply that lambda expression for each element present in the stream.

f.toArray() method

We can use toArray() method to copy elements present in the stream into specified array

g.Stream.of()method

We can also apply a stream for group of values and for arrays.

eg#1.

```
import java.util.*;
import java.util.stream.*;
class Employee
{
    int eid;
    String ename;
    float esal;
    Employee(int eid,String ename,float esal)
    {
        this.eid = eid;
        this.ename = ename;
        this.esal = esal;
    }
    public String toString()
    {
        return "["+eid+","+ename+","+esal+"]";
    }
}
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        List al = new ArrayList();
        al.add(new Employee(10,"sachin",35000f));
        al.add(new Employee(7,"dhoni",30000f));
        al.add(new Employee(18,"kohli",35000f));
        al.add(new Employee(1,"rahul",28000f));
        al.add(new Employee(19,"dravid",30000f));
        al.add(new Employee(45,"rohith",25000f));
        System.out.println(al);
        printObjectDataUsingStream(al);
    }
    public static void printObjectDataUsingStream(List al)
    {
        System.out.println();
```

```
        //Using Stream API
        long result = al.stream().filter(emp->emp.esal>30000).count();
        System.out.println(result);
        System.out.println();

        //forEach(Consumer consumer) :: void accept(T t)
        al.stream().forEach(emp->System.out.println(emp));
        System.out.println();

        //MethodReference :: instance
        al.stream().forEach(System.out::println);
        System.out.println();

        ArrayList al1 =new ArrayList();
        al1.add(0);
        al1.add(10);
        al1.add(5);
        al1.add(20);
        al1.add(15);
        System.out.println(al1);
        Integer[] arr = al1.stream().toArray(Integer[] :: new); //understand the syntax once
        for (int data:arr)
        {
            System.out.println(data);
        }
    }
}
```

Output

[[10,sachin,35000.0], [7,dhoni,30000.0], [18,kohli,35000.0], [1,rahul,28000.0], [19,dravid,30000.0], [45,rohith,25000.0]]

2

[10,sachin,35000.0]

[7,dhoni,30000.0]

[18,kohli,35000.0]

[1,rahul,28000.0]

[19,dravid,30000.0]

[45,rohith,25000.0]

[10,sachin,35000.0]

[7,dhoni,30000.0]

[18,kohli,35000.0]

[1,rahul,28000.0]

[19,dravid,30000.0]

[45,rohith,25000.0]

eg#2.

```
import java.util.*;
import java.util.stream.*;
class Employee implements Comparable
{
    Integer eid;
    String ename;
    float esal;
    Employee(Integer eid,String ename,float esal)
    {
        this.eid = eid;
        this.ename = ename;
        this.esal = esal;
    }
    public String toString()
    {
        return "["+eid+","+ename+","+esal+"]";
    }
    @Override
    public int compareTo(Object obj1)
    {
        Employee e2 = (Employee) obj1;
        int eid1 = this.eid;
        int eid2 = e2.eid;
        return eid1<eid2 ? -1 : eid1>eid2 ? +1 :0 ;
    }
}
public class Test
{
    public static void main(String[] args)
```

```
    {
        List al = new ArrayList();
        al.add(new Employee(10,"sachin",35000f));
        al.add(new Employee(7,"dhoni",30000f));
        al.add(new Employee(18,"kohli",35000f));
        al.add(new Employee(1,"rahul",28000f));
        al.add(new Employee(19,"dravid",30000f));
        al.add(new Employee(45,"rohith",25000f));
        System.out.println(al);
        printObjectDataUsingStream(al);
    }
    public static void printObjectDataUsingStream(List al)
    {
        System.out.println();
        al.stream()
        .sorted()
        .collect(Collectors.toList())
        .forEach(System.out::println);
        System.out.println();

        System.out.println("Sorting in Descending Order");
        al.stream()
        .sorted((e1,e2)->-e1.eid.compareTo(e2.eid))
        .collect(Collectors.toList())
        .forEach(System.out::println);
        System.out.println();

        Employee emp1=al.stream()
                    .min((e1,e2)->e1.esal.compareTo(e2.esal)) //no need to add - as min method will
                    handle that
                    .get();
        System.out.println("Min salary employee :: "+emp1);
        System.out.println();
        Employee emp2=al.stream()
                    .max((e1,e2)->e1.esal.compareTo(e2.esal))
                    .get();
        System.out.println("Max salary employee :: "+emp2);
    }
}
```

Output

[[10,sachin,35000.0], [7,dhoni,30000.0], [18,kohli,35000.0], [1,rahul,28000.0],

[19,dravid,30000.0], [45,rohith,25000.0]]

2

[10,sachin,35000.0]

[7,dhoni,30000.0]

[18,kohli,35000.0]

[1,rahul,28000.0]

[19,dravid,30000.0]

[45,rohith,25000.0]

[10,sachin,35000.0]

[7,dhoni,30000.0]

[18,kohli,35000.0]

[1,rahul,28000.0]

[19,dravid,30000.0]

[45,rohith,25000.0]

Object class methods

```
public class Object {
    //constructor
    public Object();

    //Commonly used in every class
    public final native Class<?> getClass();
    public native int hashCode();
    public java.lang.String toString();
    public boolean equals(Object obj);

    //Cloning
    protected native Object clone() throws CloneNotSupportedException;

    //MultiThreading
    public final native void notify();
    public final native void notifyAll();
    public final void wait() throws InterruptedException;
    public final native void wait(long) throws InterruptedException;
```

```
    public final void wait(long, int) throws InterruptedException;


    //Garbage Collector
    protected void finalize() throws java.lang.Throwable;
}
```

equals() :: check name and id,if both are equal then return true otherwise return false passing different type of objects, it would result in "ClassCastException" so return false. passing null type, it would result in "NullPointerException" so return false. if 2 references are pointing to same object, then without comparison it should return true.

Case Studies

case1:

Student s1 = new Student(10,"sachin");

Student s4 = s1;

s1==s4 :: true

s1.equals(s4) :: true

case2:

String s1 = new String("sachin");

String s2 = new String("sachin");

System.out.println(s1==s2); **//false(reference comparison)**

System.out.println(s1.equals(s2)); **//true(content comparison)**

StringBuffer s1 = new StringBuffer("sachin");

StringBuffer s2 = new StringBuffer("sachin");

System.out.println(s1==s2); **//false(reference comparison)**

System.out.println(s1.equals(s2)); **//false(object class equals() is meant for reference comparison)**

Explain the realtionsihp b/w "==" vs equals()?

1. **if r1 == r2 is true, then r1.equals(r2) will always return true.**
2. **if r1==r2 is false,then r1.equals(r2) may return true or false(depends on the reference type)**
3. if r1.equals(r2) is true,then r1==r2 may return true or false(depends on the reference type)
4. **if r1.equals(r2) is false,then r1==r2 is always false.**
5. if r1==null, then r1.equals(null) will always be false.**(comparison with null is always false - special case)**

Note:

In case of == operator,we can apply == operator on primtive types and reference type also.

In case of == operator if we apply on primitive type, it is meant for content comparison where as on reference type is meant for reference comparison.

In case of == operator, when we apply on reference type the rule is there should be a realtionship b/w both the argument, otherwise it would result in "Compiletime Error", where as in case of equals() if there is no relationship, then we wont get compiletime error rather it would return false.

Tricky Code

```
String s1 = new String("sachin");
StringBuffer sb = new StringBuffer("sachin");
System.out.println(s1==sb);  //CE
System.out.println(s1.equals(sb));   //false
```

Explain the relationship b/w equals() and hashCode() method?

```
public class Object
{
    public boolean equals(){return (this==obj);}
    public native int hashCode();
}
```

1. **if 2 objects are equal by equals() method then their hashCode must be same, it means r1.hashCode() == r2.hashCode() is true.**
2. if 2 objects are not equal by equals() method, then their hashCode may or may not be same so r1.hashCode() == r2.hashCode() can give true or false.
3. if hashcode of 2 objects are equal, then equals() may return true or false.
4. if hashcode of 2 objects are different then equals() will always return false.

Note: As a good programming practice if we are overriding equals(),compulsorily the hashCode also should be overriden. Viloation of above rule would not lead to any compiletime or runtime error,but it is not a good programming practice.

eg::

```
public int hashCode()
{
    return 100; //not a good practise
}
public int hashCode()
{
    return age+height;  //good if we use age and height in equals()
}
public int hashCode()
{
```

```
    return name.hashCode()+age; //very good,because we are use internall hashCode so hashcode will be different
}
```

eg#1.

```
class Person
{
    String name;
    int age;
    Person(String name,int age)
    {
        this.name = name;
        this.age = age;
    }
    @Override
    public int hashCode()
    {
        //implementation of hashCode: using name and age
        return name.hashCode() + age;
    }
    @Override
    public boolean equals(Object obj)
    {
        if (this == obj)
            return true;
        else if (obj instanceof Person)
        {
            //content comparison
            Person p =(Person) obj;
            if (name.equals(p.name) && age == p.age)
                return true;
            else
                return false;
        }
        return false;
    }
}
//Client Code
public class Test
```

```
{
    public static void main(String[] args)
    {
        Person p1 = new Person("sachin",51);
        Person p2 = new Person("sachin",51);
        Integer i = new Integer(100);
        System.out.println(p1.equals(p2));//true
        System.out.println(p1.equals(i));//false
    }
}
```

hashCode() => unique address generated for every object by JVM.

=> since we are talking about address, we should not touch the logic of hashCode.

=> if we are overriding the hashCode, then we need to see what properties are used in equals() method.

Note: In all String class, Collection class, Wrapper class equals() method is overriden for content comparison.

Cloning

=> The process of creating a exactly duplicate object is called "Cloning".

=> The main objective of cloning is to mantain backup.

=> To perform clonning we use "clone()" from Object class.

protected **native** Object clone() throws CloneNotSupportedException;

```
//Client Code
public class Test implements Cloneable
{
    int i =10;
    int j =20;
    Test()
    {
        System.out.println("Constructor got called");
    }
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Test t1 = new Test();
        Test t2 = (Test)t1.clone();
        t2.i = 1000;
        t2.j = 2000;
```

```
        System.out.println(t1.hashCode() == t2.hashCode());
        System.out.println("T1 Object i and j value ===>"+t1.i + " " +t1.j);//10 20
        System.out.println("T2 Object i and j value ===>"+t2.i + " " +t2.j);//1000 2000
    }
}
```

=> we can perform cloning only on "Cloneable" objects.

=> An object is said to Cloneale,if and only if the corresponding class has implemented "Cloneable" interface.

=> **Cloneable** interface is present in java.lang package and it doesn't contain any abstract method so we s**ay that interface as "marker interface".**

=> if we try to perform cloning on Non-Cloneable objects, then at runtime jvm will throw an exception called as "CloneNotSupportedExeption".

Output

Constructor got called

false

T1 Object i and j value ===>10 20

T2 Object i and j value ===>1000 2000

In java we have 2 types of Cloning

a. Shallow Cloning (+ imp diagram(13feb))

=> If the main object contains reference variable, then corresponding object won't be created rather reference will be shared to clone object, this type of clonning is called "Shallow Cloning".

=> Using the main object reference, if we perform any changes to the contianed object then those changes would be automatically reflected to main object also.

=> main object :: Dog, contained object : Cat

=> Shallow cloning is done by Object class clone() as default.

=> Shallow Cloning best suited only when the object have primitive variable types.

eg#1.

```
class Cat
{
    int j;
    Cat(int j){
        this.j = j;
    }
}
```

```
class Dog implements Cloneable
{
    int i;
    //HAS-A relationship
    Cat c;
    Dog(int i,Cat c){
        this.i = i;
        this.c = c;
    }
    //This is overriden because clone method is "protected" and such methods we can't use outside of package hence we overide it with public access modifier otherwise it will throw exception.
    @Override
    public Object clone() throws CloneNotSupportedException
    {
        //Object class clone() is getting called :: Shallow Cloning
        return super.clone();
    }
}
//Client Code
public class Test
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Cat c = new Cat(10);
        Dog d = new Dog(20,c);
        Dog d1 = (Dog) d.clone();
        //Changes are made using Cloned Copy
        d1.i = 9999;
        d1.c.j = 8888;
        System.out.println("D Object data is :: "+d.i+" "+d.c.j);
        System.out.println("D1 Object data is :: "+d1.i+" "+d1.c.j);
    }
}
```

Output

D Object data is :: 20 8888

D1 Object data is :: 9999 8888

b. Deep Cloning

If the main object contains reference variable, then corresponding object copy also will be created during clonning, such type of cloning is refered as "Deep Cloning".

Object class clone() is meant for "ShallowCloning",if we want deep cloning then we need to go for "DeepCloning".

eg#1.

```
class Cat
{
    int j;
    Cat(int j)
    {
        this.j = j;
    }
}
class Dog implements Cloneable
{
    int i;
    //HAS-A relationship
    Cat c;
    Dog(int i,Cat c){
        this.i = i;
        this.c = c;
    }
    @Override
    public Object clone() throws CloneNotSupportedException
    {
        //Perform Deep Clonning
        Cat c1 = new Cat(c.j);
        Dog d1 = new Dog(i,c1);
        return d1;
    }
}
//Client Code
public class Test
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
```

```
        Cat c = new Cat(10);
        Dog d = new Dog(20,c);
        Dog d1 = (Dog) d.clone();
        //Changes are made using Cloned Copy
        d1.i = 9999;
        d1.c.j = 8888;
        System.out.println("D Object data is :: "+d.i+" "+d.c.j);
        System.out.println("D1 Object data is :: "+d1.i+" "+d1.c.j);
    }
}
```

Output

D Object data is :: 20 10

D1 Object data is :: 9999 8888

Note:

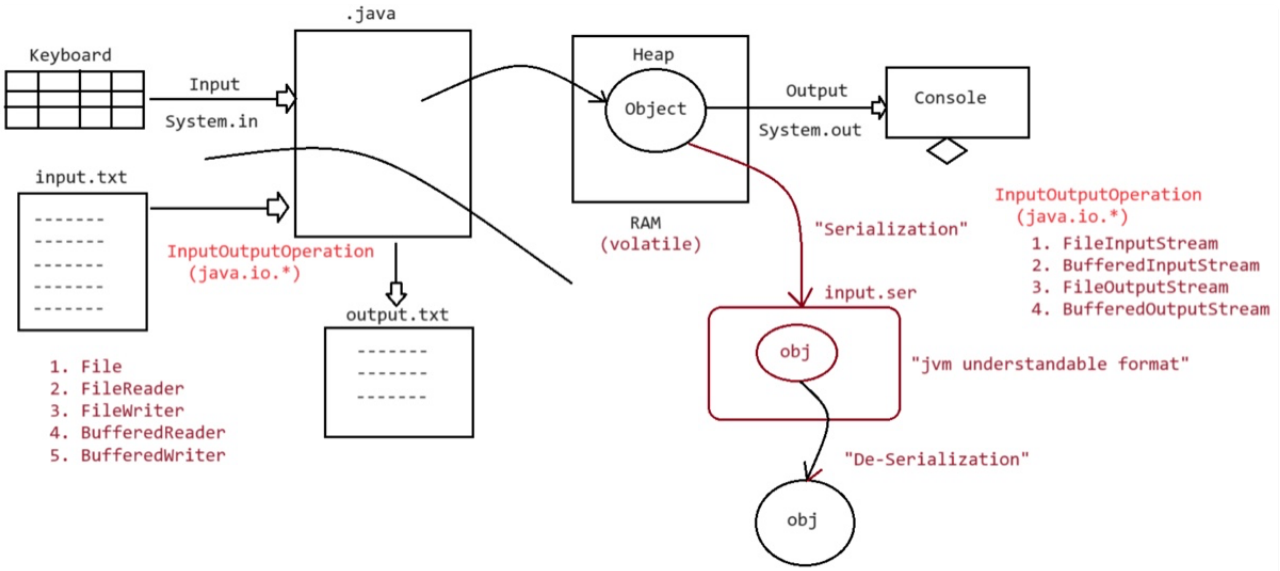
Object class clone()

a. primitive variable in object :: deep cloning

b. reference varaible in object :: Shallow cloning

c. If our object contains reference varaible and if we want deep cloning to happen then we need to override clone().

File Operations



InputOutputOperation → take data from file and after processing it again store the output in the file so that it will be available for longer run.

Default mechaninsm → keyborad to console**(data is stored in ram till the power supply is there).**

Storing java object in file → For this the format will not be in text file, so first will be converted into another format**(jvm understable format)** and the process is known as **Serialization.**

Agenda:

1. File
2. FileWriter
3. FileReader
4. BufferedWriter
5. BufferedReader

File:
File f=new File("abc.txt");
This line 1st checks whether abc.txt file is already available (or) not if it is already available then "f" simply refers that file.
If it is not already available then it won't create any physical file just creates a java File object represents name of the file.
Example:

```
import java.io.*;
class FileDemo{
    public static void main(String[] args)throws IOException{
        File f=new File("cricket.txt");
        System.out.println(f.exists());  //false
        f.createNewFile();
        System.out.println(f.exists());   //true
    }
}
```

1st run

false
true
2nd run

true. //because now the file exists

true

=> A java File object can represent a directory also.

Example:

```
import java.io.File;
import java.io.IOException;
class FileDemo{
    public static void main(String[] args)throws IOException{
```

```
        File f=new File("cricket123");  //f can point to file and folder(cricket123)
        System.out.println(f.exists());   //false
        f.mkdir();    //Creates a new directory
        System.out.println(f.exists());    //true
    }
}
```

1st run

false
true
2nd run

true

true

Note: In UNIX everything is a file, java "file IO" is based on UNIX operating system hence in java also we can represent both files and directories by File object only.

File class constructors

1. **File f=new File(String name);**
   => Creates a java File object that represents name of the file or directory in current working directory.
   eg#1. File f=new File("abc.txt");
2. **File f=new File(String subdirname,String name);**
   => Creates a File object that represents name of the file or directory present in specified sub directory.
   eg#1. File f1=new File("abc");
   f1.mkdir();
   File f2=new File("abc","demo.txt");
3. **File f=new File(File subdir,String name);**

eg#1. File f1=new File("abc");
f1.mkdir();
File f2=new File(f1,"demo.txt");

Requirement

=> Write code to create a file named with demo.txt in current working directory.
cwd
|=> demo.txt

eg#1.

import java.io.*;

//Client Code

public class Test

```
{
    public static void main(String[] args)throws IOException
    {
        //File(UNIX) :: file,folders
        File f= new File("demo.txt");
        f.createNewFile();
        System.out.println(f.exists());
    }
}
```

Requirement

=> Write code to create a directory named with IPLTeam in current working directory and create a file named with abc.txt in that directory.

```
cwd
    |=> IPLTeam
        |=> abc.txt
```

eg#1.

```
import java.io.*;
//Client Code
public class Test
{
    public static void main(String[] args)throws IOException
    {
        //File(UNIX) :: file,folders
        File f1= new File("IPLTeam");
        f1.mkdir();
        File f2 = new File(f1,"rcb.txt");
        f2.createNewFile();
        System.out.println(f1.exists());
        System.out.println(f2.exists());
    }
}
```

Requirement:

=> Write code to create a file named with rcb.txt present in D:\IPLTeam folder.

```
D
|=> IplTeam
    |-> rcb.txt
```

eg#1.

```
import java.io.*;
//Client Code
public class Test
{
    public static void main(String[] args)throws IOException
    {
        //File(UNIX) :: file,folders
        //IPLTeam already exists
        File f= new File("D:\OctBatchMicroservices\IPLTeam","rcb.txt");
        f.createNewFile();
        System.out.println(f.exists());
    }
}
```

Note: If the specified String representation path doesn't exists, then JVM will throw an Exception called "FileNotFoundException"

Important methods of file class:

1. **boolean exists();**

   Returns true if the **physical** file or directory available.

2. **boolean createNewFile();**

   This method 1st checks whether the physical file is already available or not, if it is already available then this method simply returns false without creating any physical file.

   If this file is not already available then it will create a new file and returns true

3. **boolean mkdir();**

   This method 1st checks whether the directory is already available or not if it is already available then this method simply returns false without creating any directory.

   If this directory is not already available then it will create a new directory and returns true

4. **boolean isFile();**

   Returns true if the **File object represents a physical file.**

5. **boolean isDirectory();**

   Returns true if the File object represents a directory.

6. **String[] list();**

   It returns the names of all files and subdirectories present in the specified directory.

7. **long length();**

   Returns the no of characters present in the file.

8. **boolean delete();**

   To delete a file or directory.

   true -> if directory is deleted otherwise false if directory is deleted.

Requirement: Write a program to display the names of all files and directories present in D:\EnterpriseJava

eg#1.

```
import java.io.*;
//Client Code
public class Test
{
    public static void main(String[] args)throws IOException
    {
        //File Object of java
        File f= new File("D:\EnterpriseJava");
        //Get the information of that location
        String[] info = f.list();
        int count = 0;
        //read the array and display the information
        for (String data:info )
        {
            count++;
            System.out.println(data);
        }
        System.out.println("Total no of files and directories is :: "+count);
        System.out.println("Total no of files and directories is :: "+info.length);
    }
}
```

Output

Total no of files and directories is :: 153

Requirement: Write a program to display only file names.

Requirement: Write a program to display only directory names.

eg#1

```
import java.io.*;
//Client Code
public class Test
```

```
{
    public static void main(String[] args)throws IOException
    {
        //File Object of java
        File f= new File("D:\EnterpriseJava");
        //Get the information of that location
        String[] info = f.list();
        //Count of Files
        int countOfFiles = 0;
        int countOfDirectories = 0;
        //read the array and display the information
        for (String data:info )
        {
            //Creating a File Object for "D:\EnterpriseJava"
            File f1 = new File(f,data);  //f1 will point to current file or directory
            //Checking whether File Object is "FileType"
            if (f1.isFile()){
                countOfFiles++;
                System.out.println(data);
            }
            //Checking whether File Object is "DirectoryType"
            if(f1.isDirectory())
            {
                countOfDirectories++;
                System.out.println(data);
            }
        }
        System.out.println("Total no of files is :: "+countOfFiles);
        System.out.println("Total no of Directories is :: "+countOfDirectories);
    }
}
```

Total no of files is :: 145

Total no of directories is :: 8

FileWriter:

By using FileWriter object we can write character data to the file.

Constructors:

FileWriter fw=new FileWriter(String name);

FileWriter fw=new FileWriter(File f);

The above 2 constructors meant for overriding the data to the file.

Instead of overriding if we want append operation then we should go for the following 2 constructors.

FileWriter fw=new FileWriter(String name,boolean append);

FileWriter fw=new FileWriter(File f,boolean append);

If the specified physical file is not already available then these constructors will create that file.

Methods:

1. **write(int ch);**
   To write a single character to the file.
2. **write(char[] ch);**
   To write an array of characters to the file.
3. **write(String s);**
   To write a String to the file.
4. **flush();**
   To give the guarantee the total data include last character also written to the file.
5. **close();**
   To close the stream.

eg#1.

```
import java.io.*;
public class Test
{
    public static void main(String[] args)throws IOException
    {
        //Writing character data to a file
        FileWriter fw = new FileWriter("cricketer.txt",true);
        fw.write(97);
        fw.write("PWIOI");

        char[] arr = {'P','W','S','K','I','L','L','S'};
        fw.write(arr);

        fw.flush();  //Gaurantee that data is written to a file(before this data is stored in Stream)
        fw.close();  //Closing the Stream
    }
}
```

Note:

=> The main problem with FileWriter is we have to insert line separator manually,which is difficult to the programmer. ('\n') → fw.write("/n"); /n → functionality may vary from system to system

=> And even line separator varing from system to system.

FileReader:

=> By using FileReader object we can read character data from the file.

Constructors:

FileReader fr=new FileReader(String name);

FileReader fr=new FileReader (File f);

Methods

1. **int read();**

   It attempts to read next character from the file and return its Unicode value. If the next character is not available then we will get -1.

2. **int i=fr.read();**
3. **System.out.println((char)i);**

   As this method returns unicodevalue , while printing we have to perform type casting.

4. **int read(char[] ch);**

   It attempts to read enough characters from the file into char[] array and returns the no of characters copied from the file into char[] array.

5. File f=new File("abc.txt");
6. Char[] ch=new Char[(int)f.length()];
7. void close();

eg#1.

```
import java.io.*;
//Client Code
public class Test
{
    public static void main(String[] args)throws IOException
    {
        //Reader character data from a file
        File file = new File("sample.txt");
        FileReader fr = new FileReader(file);
        //FirstAppraoch :: reading character data at a time
        int i =fr.read();
```

```
        while (i!=-1)
        {
            System.out.print((char)i);
            i = fr.read();
        }
        System.out.println();

        //SecondApproach :: reading whole file data into character array
        char[] ch = new char[(int)file.length()];
        System.out.println("No of characters in a file :: "+file.length());
        //when parameter is passed then it reads whole file and store that data in ch
        int noOfCharactersRead = fr.read(ch);
        System.out.println("OneTimeReadOperation count :: "+noOfCharactersRead);
        for (char data: ch)
        {
            System.out.print(data);
        }
        System.out.println();

        fr.close();//Closing the Stream
    }
}
```

In first approach the file access frequency is too high as it get 1 character at a time from file and print which degrades the performance.

Usage of FileWriter and FileReader is not recommended because of following reason

1. While writing data by FileWriter compulsory we should insert line separator(\n) manually which is a bigger headache to the programmer.
2. While reading data by FileReader we have to read character by character instead of line by line which is not convenient to the programmer. Assume we need to search for a 10 digit mobile no present in a file called "mobile.txt"

   =>Since we can read only character just to search one mobile no 10 searching and to search 10,000 mobile no we need to read 1cr times, so performance is very low.
3. **To overcome these limitations we should go for BufferedWriter and BufferedReader concepts.**