TEKNOTURF

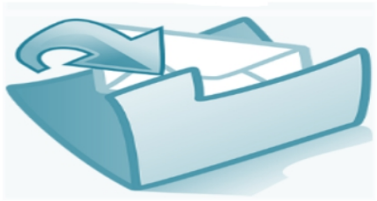
COLLECTION

Collection

TEKNOTURF

Collection is a container that groups multiple elements into a single unit where each element is an object

Collection is used to store, retrieve and manipulate aggregate data



Collection Framework

TEKNOTURF

All classes and interfaces related to Collection are present in java.util package

Collection framework

- Is a set of utility class and interfaces.
- Designed for working with collection of objects.
- Will hold objects

Collection framework contains

- Set of interfaces
- Classes that implement those interfaces
- Utility classes for performing operations on collections

Types of Collection

TEKNOTURF

Simple Collection

Sets have no duplicate elements.

Lists are ordered collections that can have duplicate elements.

Map

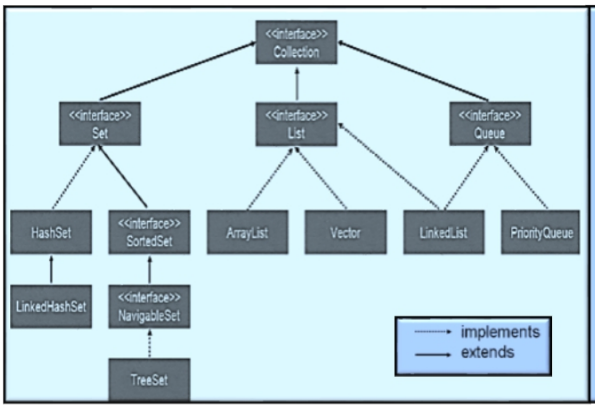
Map uses key/value pairs to associate an object (the value) with a key.

Both set and map can be sorted or unsorted.

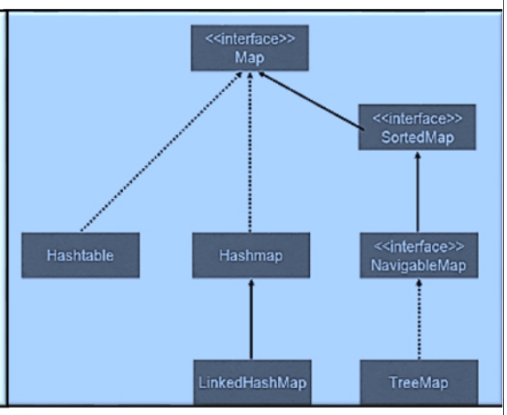
Collection Hierarchy

TEKNOTURF

Collection Hierarchy



Map Hierarchy



Collection – Implementing Classes

TEKNOTURF

There are several general purpose implementations of the core interfaces

(Set, List, Deque and Map)

	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

List Interface

Used for storing elements in an ordered way

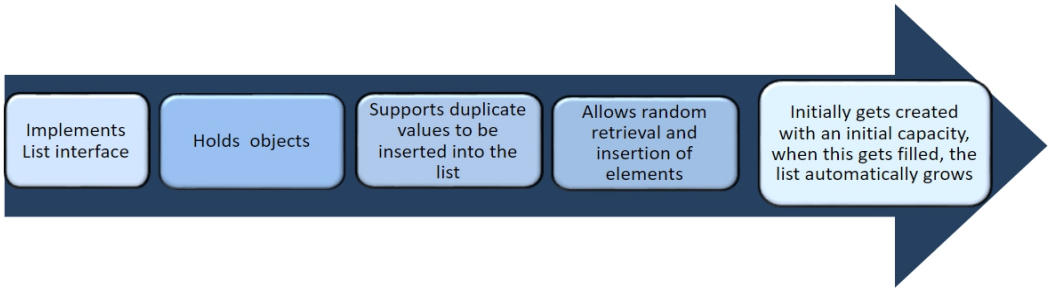
Allows index based additions and retrieval of items

Allows duplicate elements

List Implementations

- Vector
- ArrayList
- LinkedList

ArrayList



ArrayList Methods

Method	Description
void add(int index, Object element)	Inserts the specified element at the specified position in this list.
boolean add(Object o)	Appends the specified element to the end of this list.
boolean addAll(Collection)	Appends all of the elements in the specified Collection to the end of this list, in the order that they are returned by the specified Collection's Iterator.
boolean addAll(int index, Collection c)	Inserts all of the elements in the specified Collection into this list, starting at the specified position.
void clear()	Removes all of the elements from this list.
boolean contains(Object elem)	Returns true if this list contains the specified element.
Object get(int index)	Returns the element at the specified position in this list.

More methods refer to:
https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

List Example

```
import java.util.*;
public class ListExample {
        public static void main(String a[])
        {
            List list = new ArrayList();
            list.add("First");
            list.add("2nd");
            list.add("third");
            list.add(new Float(4.0f));
            list.add(new Integer(6));
             list.add(5);
            list.add("third");                  //duplicate added
            list.add(5);                        //duplicate added
            System.out.println(list);
        }
}
```

Output : [First, 2nd, third, 4.0, 6,5, third, 5]

Set Interface

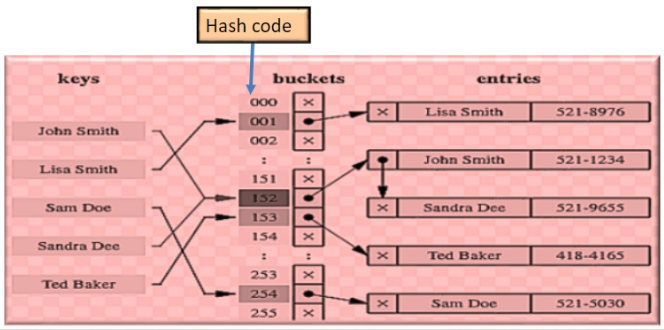
- A Set is a collection with no duplicate elements.
- A HashSet stores elements in a hash table.
- A TreeSet stores elements in a balanced binary tree.
- A LinkedHashSet is an ordered hash table.

HashSet

HashSet is a powerful data structure that can be used to retrieve objects quickly in a set.

HashSet is an Unordered collection

A hash code is used to organize the objects in a HashSet



HashSet Methods

Method	Description
boolean add(Object element)	Adds the specified element to this set if it is not already present.
void clear()	Removes all of the elements from this set.
boolean contains(Object o)	Returns true if this set contains the specified element.
boolean isEmpty()	Returns true if this set contains no elements.
boolean remove(Object o)	Removes the specified element from this set if it is present.
int size()	Returns the number of elements in this set.

- More methods refer to:
  https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

TreeSet

Tree Set provides the properties of a set in a sorted collection

Objects can be inserted in any order but are retrieved in a sorted order

TreeSet uses Comparable interface to compare objects for sorting

It is up to the programmer to implement the compareTo() method for user-defined objects

A Set Example

```
import java.util.*;
public class SetExample {
        public static void main(String a[])
        {
            Set set = new HashSet();
            set.add("First");
            set.add("2nd");
            set.add("third");
            set.add(new Float(4.0f));
            set.add(new Integer(6));
            set.add(5);
            set.add("third");              //duplicate, not added
            set.add(5);   //duplicate, not added
            System.out.println(set);
        }
}
```

- The output generated from this program is: [4.0, third, 5, 6, 2nd, First]

Map Interface

Map are sometimes called associative arrays

Example : Dictionary

A Map object describes mappings from keys to values:

- Duplicate keys are not allowed
- One-to-many mappings from keys to values are not permitted

The contents of the Map interface can be viewed and manipulated as collections

- entrySet – Returns a Set of all the key-value pairs.
- keySet – Returns a Set of all the keys in the map.
- values – Returns a Collection of all values in the map.

Map implementation

TreeMap orders the keys.

HashMap stores the keys in a hash table.

HashMap class is generally preferred for its efficiency (speed) unless sorted keys are needed.

MAP INTERFACE METHODS

Methods	Description
void clear()	Removes all mappings from this map (optional operation).
boolean containsKey(Object key)	Returns true if this map contains a mapping for the specified key.
boolean containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
Set entrySet()	Returns a set view of the mappings contained in this map.
Object get(Objectkey)	Returns the value to which this map maps the specified key.
Set keySet()	Returns a set with all the keys contained in this map.
Object put(Object key, Objectvalue)	Associates the specified value with the specified key in this map.
void putAll(Map t)	Copies all of the mappings from the specified map to this map
Object remove(Object key)	Removes the mapping for this key from this map if it is present
int size()	Returns the number of key-value mappings in this map.
Collection values()	Returns a collection with all the values contained in this map.

A Map Example


```
import java.util.*;
public class  MapExample {
        public static void main(String a[])  {
            Map map=new HashMap();
            map.put("First","1st");
            map.put("2nd",new Float(2.0f));
            map.put("third","3rd");
            //Duplicate - Overrides the previous assignment
            map.put("third","III");

            //To view the map
            //Returns the set view of keys
            Set set=map.keySet();
            //Returns collection view of values
            Collection collection = map.values();
            // Returns set view of key value mappings
            Set mapset=map.entrySet();
            System.out.println(set+"\n"+collection+"\n"+mapset);
        }
}
```

Output :

[2nd, First, third]

[2.0, 1st, III]

[2nd=2.0, First=1st, third=III]

Sorted Map

SortedMap is similar to SortedSet, but it holds the elements as key-value pair

- The key will be in a sorted manner
- Implementation classes: TreeMap
- TreeMap: Key Objects can be inserted in any order but are retrieved in a sorted order
- TreeMap internally uses compareTo() methods to compare the key objects for sorting.

Iterating Collections

TEKNOTURF

Iteration is the process of retrieving every element in a collection

The basic iterator interface allows to scan forward through any collection

For iterating a list, we can use the ListIterator which allows to scan a list both forward and backward and insert or modify elements

Iterating Collections

TEKNOTURF

Following are the ways to iterate the collection and fetch elements one by one

for loop

- Can be used only with List
- Reads the elements by specifying the index using the get() method

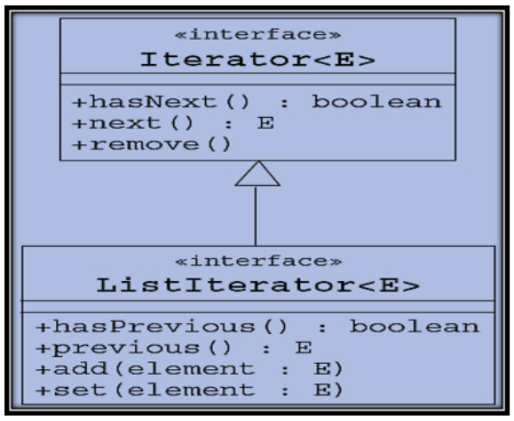
for each loop

- Can be used with both List and Set
- Iterates and fetches the element one by one until all the elements in the list are covered

Iterator

- Can be used with both List and Set
- Provides methods to iterate over the collection

Iterator interface



Using for, foreach and Iterator

```
List<Student> studentList = new
                        ArrayList<Student>();
//add some elements
for(int i=0;i<studentList.size();i++)  {
      System.out.println(studentList.get(i));
}
```

```
List<Student> studentList = new
                        ArrayList<Student>();
//add some elements
for(Student  studentObj : studentList)  {
      System.out.println(studentObj);
}
```

```
List<Student> studentList = new ArrayList<Student>();
//add some elements
Iterator<Student> elements = studentList.iterator();
while(elements.hasNext()) {
      Student studentObj=elements.next();
      System.out.println(studentObj);
}
```

Iteration using For loop

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

   public static void main(String a[])
   {
       List list = new ArrayList();
       list.add("First");
       list.add("2nd");
       list.add("third");
       list.add(new Float(4.0f));
       list.add(new Integer(5));
       list.add("third");
       list.add(5);
       for(int i=0;i<list.size();i++)
       {
           System.out.println(list.get(i));
       }

   }
}
```

The list.size() will return the value as 7

The get method takes the index position and fetches the object in that location

Iteration using For loop(with typecasting)

Program to print the string values, if the length of the string is 5

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

  public static void main(String a[])
  {
      List list = new ArrayList();
      list.add("java");
      list.add("oracle");
      list.add("mysql");
      list.add("database");
      list.add("datamining");

      for(int i=0;i<list.size();i++)
      {
          String name=list.get(i);   Type mismatch: cannot convert from Object to String
          if(name.length()==5)
          {
              System.out.println(name);
          }
      }
  }
}
```

Why is there an error in the line
String name=list.get(i)

The get method will return an Object, it's the responsibility of the programmer to type cast to the corresponding object.

We need to typecast as
String name=(String)list.get(i);

Iteration using enhanced for loop

Program to print the string values, if the length of the string is 5

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

    public static void main(String a[])
    {
        List list = new ArrayList();
        list.add("java");
        list.add("oracle");
        list.add("mysql");
        list.add("database");
        list.add("datamining");

        for(Object obj:list)
        {
            String name=(String)obj;
            if(name.length()==5)
            {
                System.out.println(name);
            }
        }
    }
}
```

Enhanced for loop, is used for fetching each object. It is then type casted and displayed if the length is 5

Iteration using iterator

Program to print the string values, if the length of the string is 5

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Test {
  public static void main(String a[])
  {
      List list = new ArrayList();
      list.add("java");
      list.add("oracle");
      list.add("mysql");
      list.add("database");
      list.add("datamining");

      Iterator iobj=list.iterator();
      while(iobj.hasNext())
      {

          String name=(String)iobj.next();
          if(name.length()==5)
          {
              System.out.println(name);
          }
      }
  }
}
```

The iterator() method returns an iterator over the elements in the ArrayList.

Iteration using iterator

Program to print the string values, if the length of the string is 5

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Test {
  public static void main(String a[])
  {
      List list = new ArrayList();
      list.add("java");
      list.add("oracle");
      list.add("mysql");
      list.add("database");
      list.add("datamining");

      Iterator iobj=list.iterator();
      while(iobj.hasNext())
      {
          String name=(String)iobj.next();
          if(name.length()==5)
          {
              System.out.println(name);
          }
      }
  }
}
```

The iterator() method returns an iterator over the elements in the ArrayList.

In the while loop, we check whether the iterator has more elements by invoking the hasNext() method

Iteration using iterator

Program to print the string values, if the length of the string is 5

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Test {
  public static void main(String a[])
  {
      List list = new ArrayList();
      list.add("java");
      list.add("oracle");
      list.add("mysql");
      list.add("database");
      list.add("datamining");

      Iterator iobj=list.iterator();
      while(iobj.hasNext())
      {
          String name=(String)iobj.next();
          if(name.length()==5)
          {
              System.out.println(name);
          }
      }
  }
}
```

The iterator() method returns an iterator over the elements in the ArrayList.

In the while loop, we check whether the iterator has more elements by invoking the hasNext() method

Using next() we fetch the object from the iterator, this method returns an object , it is then type casted and displayed if the length is 5

Iteration using iterator

To iterate the elements in a HashMap

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class Directory{

    public static void main(String[] args) {

        Map map = new HashMap();
        map.put("Alan", "9843012345"); //Key, Value pair
        map.put("Scott", "7780521155");
        map.put("Thom", "8002598765");

        //Iterating using Iterator
        Iterator iter = map.entrySet().iterator();
        while (iter.hasNext()) {
            Map.Entry entry = (Map.Entry) iter.next();
            System.out.println(entry.getKey() + " - " + entry.getValue());
        }
    }
}
```

In Map, Iteration is done using entrySet().iterator() method; It helps in iterating the elements of the collection

Created a HashMap of Names and Phone numbers

Names are the Keys and Phone Numbers are the values

Output:

```
Thom - 8002598765
Alan - 9843012345
Scott - 7780521155
```

foreach method

This method was introduced in java 8

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

public class Test {

  public static void main(String[] args) {
      List list = new ArrayList();
      list.add("java");
      list.add("oracle");
      list.add(10.6);
      list.add("mysql");
      list.add(123);

      list.forEach(s->System.out.println(s));
  }
}
```

-> is a lambda expression

foreach method

This method was introduced in java 8

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

public class Test {

  public static void main(String[] args) {
      List list = new ArrayList();
      list.add("java");
      list.add("oracle");
      list.add(10.6);
      list.add("mysql");
      list.add(123);

      list.forEach(s->System.out.println(s));
  }
}
```

-> is a lambda expression

forEach() method is defined in Iterable and Stream interface.

It is a default method in the Iterable interface.

Collection classes which extends Iterable interface can use forEach loop to iterate elements.

Method : void forEach(Consumer<super T>action)

- Performs the given action for each element of the Iterable until all the elements have been processed

For more methods refer to the link:: https://docs.oracle.com/en/java/javase

foreach method

Example to iterate over map

```
import java.util.HashMap;
import java.util.Map;
class Customer
{
    int id;
    String name;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Customer(int id, String name) {
        super();
        this.id = id;
        this.name = name;
    }
}
```

```
public class StreamDemo {

    public static void main(String[] args) {

    Map<Integer, Customer> map = new HashMap<Integer,Customer>();
    map.put(1, new Customer(12, "peter"));
    map.put(2, new Customer(34, "john"));
    map.put(3, new Customer(34, "tom"));
    map.forEach((k, v) -> System.out.println("Key : " + k + " Value : " + v.getId()));

    }

}
```

GENERICS

TEKNOTURF

Introduced from Java 5

Allows the programmer to specify the datatype to be stored in the collection

Provides compile-time type safety

Eliminates the need for casts

Provides the ability to create compiler-checked homogeneous collections

Using non-generic collections :

```
ArrayList list = new ArrayList();
list.add(0,new Integer(42));
int total = ((Integer)list.get(0))
```

Using generic collections :

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0,new Integer(42));
int total = list.get(0);
```

Generic classes

TEKNOTURF

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas.

These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Generic Class contd.

Example–

```
public class Box<T> {
        private T t;
        public void add(T t) {
            this.t = t;
        }
        public T get() {
            return t;
        }
        public static void main(String[] args) {
            Box<Integer> integerBox = new Box<Integer>();
            Box<String> stringBox = new Box<String>();
            integerBox.add(new Integer(10));
            stringBox.add(new String("Hello World"));
            System.out.println("Integer Value : " + integerBox.get());
            System.out.println("String Value : "+stringBox.get());
        }
}
```

Output :

Integer Value : 10
String Value : Hello World

Generic class using Type Parameters

Classes showing how to use type parameters

Category	Non Generic Class	Generic Class
Class declaration	`public class ArrayList extends AbstractList implements List`	`public class ArrayList<E> extends AbstractList<E> implements List <E>`
Constructor declaration	`public ArrayList (int capacity);`	`public ArrayList<E> (int capacity);`

Method declaration	`public void add((Object o)`
	
	`public Object get(int index)`
		`public void add(E o)`
		
		`public E get(int index)`

Variable declaration examples	`ArrayList list1;`
	
	`ArrayList list2;`
		`ArrayList <String> list1;`
		
		`ArrayList <Date> list2;`

Instance declaration examples	`list1 = new ArrayList(10);`
	
	`list2 = new ArrayList(10);`
		`list1= new ArrayList<String> (10);`
		
		`list2= new ArrayList<Date> (10);`

Generics with Collection

```
import java.util.*;
public class SetExample {
        public static void main(String a[])
        {
            Set<String> set = new HashSet<String>();
            set.add("First");
            set.add("2nd");
            set.add("third");
            //This line generates a compilation error
                    //  set.add(new Float(4.0f));
            //This line generates a compilation error
                    //  set.add(5);
            set.add("third");               //duplicate, not added
            System.out.println(set);
          }
}
```

Generics with Collection

```
public class PhoneBook {

    HashMap<String,String> phoneBook;

    public PhoneBook() {
        phoneBook = new HashMap<String,String>();
    }

    public void put(String name,String phno) {
        phoneBook.put(name,phno);
    }

    public String get(String name){
        return phoneBook.get(name);
    }
}
```

```
public class ListExample  {
    public static void main(String a[])
    {
        PhoneBook ph=new PhoneBook();
        String name="Peter";
        ph.put("John","9994441231");
        ph.put("Peter","9994441232");
        ph.put("James","9994441233");
        System.out.println("Phone Number of
                "+name+" is "+ph.get("Peter"));
    }
}
```

Output :
Phone Number of Peter is 9994441232

ArrayList for User defined class

```
public class Employee {
    private int id;
    private String name;

//Write the Getters, Setters and Constructor
        @Override
    public String toString()  {
        return "ID : "+id+" Name : "+name;
    }
}
```

Output :

ID : 101 Name : Tina

ID : 109 Name : Pooja

ID : 122 Name : George

ID : 101 Name : Peter

```
import java.util.*;

public class ArrayListDemo {

    public static void main(String[] args) {
        Employee e1 = new Employee(101,"Tina");
        Employee e2 = new Employee(109,"Pooja");
        Employee e3 = new Employee(122,"George");
        Employee e4 = new Employee(101,"Peter");
        List<Employee> empList=new ArrayList<Employee>();
        empList.add(e1);
        empList.add(e2);
        empList.add(e3);
        empList.add(e4);
        for(Employee e : empList)
            System.out.println(e);
    }

}
```

equals() and hashcode()

- public boolean equals(Object obj) - A method of Object class and checks if the object passed as an argument is equal to the current instance.
- public int hashcode() - A method of Object class, that returns an integer value represented by a hashing function.
  By default the hashCode() returns a random integer that is unique for each instance or object.
- For HashSet or HashMap, overriding both hashCode() and equals() methods are required.
  Two objects having the same hash code does not imply that they are equal.
  But two objects that are equal will have the same hashCode.

```
@Override
public int hashCode()
{
  //some code
}
```

```
@Override
public boolean equals(Object obj)
{
      //some code
}
```

HashSet for User defined class

Example for HashSet with User Defined class

```
public class Employee{

        private int id;
        private String name;

       //Assume you have written getters, setters and constructor

       @Override
       public int hashCode()   {
           return id%7;
       }
       @Override
       public boolean equals(Object o) {
           Employee e=(Employee)o;
           if(this.id == e.id)
               return true;
           else
               return false;
       }

       @Override
       public String toString() {
           return "Id = " + id + ", Name = " + name ;
       }
}
```

Assume there is a class Employee with id, name and age as attributes. Also assume the constructor, getters and setters are written

HashSet for User defined class

Example for HashSet with User Defined class

```
public class Employee{

        private int id;
        private String name;

       //Assume you have written getters, setters and constructor

       @Override
       public int hashCode()   {
           return id%7;
       }
       @Override
       public boolean equals(Object o) {
           Employee e=(Employee)o;
           if(this.id == e.id)
               return true;
           else
               return false;
       }

       @Override
       public String toString() {
           return "Id = " + id + ", Name = " + name ;
       }
}
```

For the user defined object, it is the responsibility of the programmer to override the hashCode() and equals() method to avoid duplicates.

HashSet for User defined class

Example for HashSet with User Defined class

```
public class Employee{

        private int id;
        private String name;

       //Assume you have written getters, setters and constructor

       @Override
       public int hashCode()   {
           return id%7;
       }
       @Override
       public boolean equals(Object o) {
           Employee e=(Employee)o;
           if(this.id == e.id)
               return true;
           else
               return false;
       }

       @Override
       public String toString() {
           return "Id = " + id + ", Name = " + name ;
       }
}
```

When we add elements to hashset, it generates the hashcode for each object.

And if any object exist with the same code, it invokes the equals method.

If the equals method returns true, the object will not be added, else it will be added to the hashset.

HashSet for User defined class

Example for HashSet with User Defined class

```
import java.util.HashSet;
import java.util.Set;

public class HashSetDemo {

    public static void main(String[] args) {

        Employee employeeObj1 = new Employee(101,"Tina");   //hash code is 3
        Employee employeeObj2 = new Employee(109,"Pooja"); //hash code is 4
        Employee employeeObj3 = new Employee(122,"George"); // hash code is 3,
                                                //id not matching. so added
        Employee employeeObj4 = new Employee(101,"Peter"); //hash code is 3,
                                                //id matching. so not added

        Set<Employee> empList=new HashSet<Employee>();
        empList.add(employeeObj1);
        empList.add(employeeObj2);
        empList.add(employeeObj3);
        empList.add(employeeObj4);

        for(Employee empObj : empList)  {
            System.out.println(empObj.getId());
            System.out.println(empObj.getName());
        }

    }
}
```

For the employeeObj1, the hash code is computed as 101 % 7 which is 3;
Tina is added as it is the first object and
Does not need to invoke equals() method to check duplicates

HashSet for User defined class

Example for HashSet with User Defined class

```
import java.util.HashSet;
import java.util.Set;

public class HashSetDemo {

    public static void main(String[] args) {

        Employee employeeObj1 = new Employee(101,"Tina");   //hash code is 3
        Employee employeeObj2 = new Employee(109,"Pooja"); //hash code is 4
        Employee employeeObj3 = new Employee(122,"George"); // hash code is 3,
                                                //id not matching. so added
        Employee employeeObj4 = new Employee(101,"Peter"); //hash code is 3,
                                                //id matching. so not added

        Set<Employee> empList=new HashSet<Employee>();
        empList.add(employeeObj1);
        empList.add(employeeObj2);
        empList.add(employeeObj3);
        empList.add(employeeObj4);

        for(Employee empObj : empList)  {
            System.out.println(empObj.getId());
            System.out.println(empObj.getName());
        }

    }
}
```

For the employeeObj2, the hash code is computed as 109 % 7 which is 4;
Pooja is added as the hash code is different and does not need to invoke equals() method to check duplicates

HashSet for User defined class

Example for HashSet with User Defined class

```
import java.util.HashSet;
import java.util.Set;

public class HashSetDemo {

    public static void main(String[] args) {

        Employee employeeObj1 = new Employee(101,"Tina");   //hash code is 3
        Employee employeeObj2 = new Employee(109,"Pooja"); //hash code is 4
        Employee employeeObj3 = new Employee(122,"George"); // hash code is 3,
                                                //id not matching. so added
        Employee employeeObj4 = new Employee(101,"Peter"); //hash code is 3,
                                                //id matching. so not added

        Set<Employee> empList=new HashSet<Employee>();
        empList.add(employeeObj1);
        empList.add(employeeObj2);
        empList.add(employeeObj3);
        empList.add(employeeObj4);

        for(Employee empObj : empList)  {
            System.out.println(empObj.getId());
            System.out.println(empObj.getName());
        }

    }
}
```

For the employeeObj3, the hash code is computed as 122 % 7 which is 3; As 3 is already added, the HashSet also invokes equals() method to check duplicates on id. But since 122 and 101 are different, George is added

HashSet for User defined class

Example for HashSet with User Defined class

```
import java.util.HashSet;
import java.util.Set;

public class HashSetDemo {

    public static void main(String[] args) {

        Employee employeeObj1 = new Employee(101,"Tina");   //hash code is 3
        Employee employeeObj2 = new Employee(109,"Pooja"); //hash code is 4
        Employee employeeObj3 = new Employee(122,"George"); // hash code is 3,
                                                //id not matching. so added
        Employee employeeObj4 = new Employee(101,"Peter"); //hash code is 3,
                                                //id matching. so not added

        Set<Employee> empList=new HashSet<Employee>();
        empList.add(employeeObj1);
        empList.add(employeeObj2);
        empList.add(employeeObj3);
        empList.add(employeeObj4);

        for(Employee empObj : empList)  {
            System.out.println(empObj.getId());
            System.out.println(empObj.getName());
        }

    }
}
```

For the employeeObj4, the hash code is computed as 101 % 7 which is 3; As hash code 3 already exists, the HashSet also invokes equals() method to check duplicates on id. But 101 id is duplicate, and so Peter is not added

Comparable and Comparator

Comparable and Comparator are used for sorting collection of objects.

To apply sorting methods of Arrays or Collections on any user defined class, that class should implement the Comparable interface.

Comparable interface has **compareTo(Tobj)** method. This method should be overridden by the class implementing Comparable to sort the objects in the collection.

Comparator interface has *compare(Object o1, Object o2)* method which needs to be implemented by the class to sort the collection of objects.

Comparable Vs Comparator

Comparable	Comparator
We can sort the collection of objects on the basis of a single element. Say, by id or by name or by price.	We can sort the collection on the basis of multiple elements such as id, name, price etc.
Comparable affects the original class.	Comparator doesn't affect the original class.
Comparable provides compareTo() method to sort elements.	Comparator provides compare() method to sort elements.

Comparable Example

```
public class Employee implements Comparable<Employee> {

    private int id;
    private String name;
    private int age;
    //Assume you have written Getters, Setters and Constructor

    public int compareTo(Employee obj) {

        if(this.id==obj.id)
            return 0;
        else if(this.id > obj.id)
            return 1;
        else
            return -1;
    }

    public String toString()    {
        return "ID : "+id+" Name : "+name+" Age"+age;
    }

}
```

For the user defined object, it's the responsibility of the programmer to implement the Comparable interface and override the compareTo() method to store the elements in the sorted manner.

TreeSet Example

```
import java.util.TreeSet;

public class TreeSetDemo {

        public static void main(String[] args) {

            TreeSet<Employee> empList = new TreeSet<Employee>();

            empList.add(new Employee(107,"George"));
            empList.add(new Employee(105,"John"));
            empList.add(new Employee(102,"Tom"));
            empList.add(new Employee(105,"Peter")); //not added

            for(Employee empObj : empList){

                System.out.println(empObj);

            }
        }
    }
```

In the example, when we add the Employee objects into the TreeSet, the compareTo method will be invoked.

If the employee id is same it will return 0. In that case that object will not be added to the TreeSet thus restricting duplicates.

If the method returns 1 or -1 the elements will be sorted either in the ascending or the descending order.

Comparator

The Comparator interface is used when there is a need to sort objects in an order other than their natural ordering

The Comparator is also used for user defined objects like Employee, Customer or Student which does not have a natural ordering or does not implement Comparable

Like the Comparable interface, the Comparator interface also has a single method.

```
public interface Comparator<T> {
            int compare(T object1, T object2);
    }
```

Comparator Example

TEKNOTURF

To compare age - Ascending

```
public class Employee {

    private int id;
    private String name;
    private int age;

    //Assume you have written Getters, Setters and Constructor
    public String toString()    {       +" Age"+age;
        return "ID : "+id+" Name : "+name;
    }
}
```

```
import java.util.Comparator;

public class AgeAscendingComparator implements Comparator<Employee>
{
    public int compare(Employee emplObj1, Employee emplObj2)
    {
        return emplObj1.getAge() - emplObj2.getAge();
    }
}
```

To compare name

```
import java.util.Comparator;

public class NameComparator implements Comparator<Employee>
{
    public int compare(Employee emplObj1, Employee emplObj2)
    {
        return emplObj1.getName().compareTo(emplObj2.getName());
    }
}
```

To compare age - Descending

```
import java.util.Comparator;

public class AgeDescendingComparator implements Comparator<Employee>
{
    public int compare(Employee emplObj1, Employee emplObj2)
    {
        return emplObj2.getAge() - emplObj1.getAge();
    }
}
```

Comparator Example

TEKNOTURF

```
import java.util.*;

public class EmployeeListDemo{

    public static void main(String[] args)  {

        List<Employee> empList = new ArrayList<Employee>();
        Employee employeeObj1=new Employee(101, "John", 28);
        Employee employeeObj2=new Employee(109, "Peter", 25);
        Employee employeeObj3=new Employee(102, "Alan", 25);
        empList.add(employeeObj1);
        empList.add(employeeObj2);
        empList.add(employeeObj3);

        Collections.sort(empList, new NameComparator());
        System.out.println(empList);

        Collections.sort(empList, new AgeAscendingComparator());
        System.out.println(empList);

        Collections.sort(empList, new AgeDescendingComparator());
        System.out.println(empList);

    }
}
```

Output

Name Comparator
[Id : 102 Name : Alan Age : 35, Id : 101 Name : John Age : 28, Id : 109 Name : Peter Age : 25]

Age Ascending
[Id : 109 Name : Peter Age : 25, Id : 101, Name : John Age : 28, Id : 102 Name : Alan Age : 35]

Age Descending
[Id : 102 Name : Alan Age : 35, Id : 101 Name : John Age : 28, Id : 109 Name : Peter Age : 25]

Comparator Example

TEKNOTURF

```
public class Employee {

    private int id;
    private String name;
    private double salary;


    //Assume you have written Getters, Setters and Constructor
    public String toString()    {
        return "ID : "+id+" Name : "+name+" Salary: "+salary;
    }
}
```

To compare name

```
import java.util.Comparator;

public class NameComparator implements Comparator<Employee>
{
    public int compare(Employee emplObj1, Employee emplObj2)
    {
        return emplObj1.getName().compareTo(emplObj2.getName());
    }
}
```

To compare salary

```
import java.util.Comparator;

public class SalaryComparator implements Comparator<Employee>
{
    public int compare(Employee emplObj1, Employee emplObj2)
    {
        return Double.valueOf(emplObj1.getSalary()).compareTo
                                        (emplObj2.getSalary());
    }
}
```

In the example, we have written two comparator classes, one to compare based on the name and another to compare based on the salary.

Comparator Example

TEKNOTURF

```
import java.util.*;

public class EmployeeListDemo{

    public static void main(String[] args)  {

        List<Employee> empList = new ArrayList<Employee>();
        Employee employeeObj1=new Employee(107, "John", 20000);
        Employee employeeObj2=new Employee(105, "John", 30000);
        Employee employeeObj3=new Employee(109, "Tom", 40000);
        Employee employeeObj4=new Employee(108, "John", 25000);
        empList.add(employeeObj1);
        empList.add(employeeObj2);
        empList.add(employeeObj3);
        empList.add(employeeObj4);


        TreeSet<Employee> empListSortedByName = new TreeSet<>(new NameComparator());
        empListSortedByName.addAll(empList);
        for(Employee empObj : empListSortedByName)
        System.out.println(empListSortedByName);

        System.out.println("*****************************************");
        TreeSet<Employee> empListSortedBySalary = new TreeSet<>(new SalaryComparator());
        empListSortedBySalary.addAll(empList);
        for(Employee empObj : empListSortedByName)
            System.out.println(empListSortedByName);

    }
}
```

OUTPUT:

```
ID : 107 Name : John Salary : 20000.0
ID : 109 Name : Tom Salary : 40000.0
**********
ID : 107 Name : John Salary : 20000.0
ID : 108 Name : John Salary : 25000.0
ID : 105 Name : John Salary : 30000.0
ID : 109 Name : Tom Salary : 40000.0
```

Map Example

TEKNOTURF

```
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Set;

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

        Map<String,String> map=new LinkedHashMap<>();

        map.put("Rahul", "9878787655");
        map.put("Raghu", "8971231233");
        map.put("Pooja", "7891234567");

        Set<String> keys = map.keySet();

        for(String  name : keys)  {
            System.out.println("Key is "+name);
            System.out.println("Value is "+map.get(name));
        }

        for(Map.Entry<String, String> entrySet : map.entrySet())  {
            System.out.println("Key is "+entrySet.getKey());
            System.out.println("Value is "+entrySet.getValue());
        }

        if(map.containsKey("Rahul"))  {
            map.put("Rahul", map.get("Rahul")+","+"7893455675");
        }
    }
}
```

Generics AND POLYMORPHISM

TEKNOTURF

Generic collections gives the benefits of type safety

Assume the Student class is an abstract class.

Class Hosteller and DayScholar classes are derived from Student.

```
public class Student {
    private int id;
    private String name;
    private double tutionFees;

    public Student(int id, String name,double tutionFees) {
        this.id = id;
        this.name = name;
        this.tutionFees=tutionFees;
    }

    //Assume Getters Setters are written

    public String toString(){
        return "ID "+id+" Name "+name+" Tution Fees "+tutionFees;
    }
}
```

```
public class DayScholar extends Student  {

    private double transportFees;

    public DayScholar(int id, String name, double tutionFees, int transportFees) {
        super(id, name,tutionFees);
        this.transportFees = transportFees;
    }

    public double calculateTotalFees(){
        return getTutionFees()+transportFees;
    }

    public String toString()
    {
        return super.toString()+" Transport Fees "+transportFees;
    }

}
```

Generics AND POLYMORPHISM

List<Student> studList = new ArrayList<Student>(); can accommodate any Student object getting added which can be a Student or a Hosteller or DayScholar.object

```
public class Hosteller extends Student {

    private double hostelFees;
    private double messFees;

    public Hosteller(int id, String name, double tutionFees,
            double hostelFees,double messFees) {
        super(id, name,tutionFees);
        this.hostelFees = hostelFees;
        this.messFees=messFees;
    }

    public double calculateTotalFees(){
        return getTutionFees()+hostelFees+messFees;
    }

    public String toString()
    {
        return super.toString()+" Hostel Fees "+
                        hostelFees+" Mess Fees "+messFees;
    }
}
```

```
import java.util.*;

public class StudentDAO {

    List<Student> studentList = new ArrayList<Student>();

    public void addStudent(Student studentObj){
        studentList.add(studentObj);
    }
    public List<Student> getStudentList() {
        return studentList;
    }
    public void setStudentList(List<Student> studentList) {
        this.studentList = studentList;
    }
}
```

Generics AND POLYMORPHISM

```
public class Main {

    public static void main(String[] args) {

        Student studentObj1=new Student(101,"Raghu",75000);

        Hosteller hostellerObj1=new Hosteller(102,"Dravid",
                                    80000,40000,20000);

        DayScholar dayScholarObj1=new DayScholar(103,"Raghu",
                                                90000,40000);
        StudentDAO dao = new StudentDAO();

        dao.addStudent(studentObj1);
        dao.addStudent(dayScholarObj1);
        dao.addStudent(hostellerObj1);

        System.out.println(dao.getStudentList());
    }
}
```

Output :

[ID 101 Name Raghu Tution Fees 75000.0,
ID 103 Name Raghu Tution Fees 90000.0 Transport Fees 40000.0,
ID 102 Name Dravid Tution Fees 80000.0 Hostel Fees 40000.0 Mess Fees 20000.0]

Generics AND POLYMORPHISM

Observe the below code

```
import java.util.*;

public class StudentDAO {

    List<Student> studentList = new ArrayList<Student>();

    public void addStudentList(List<Student> studentList){
        studentList.addAll(studentList);
    }
    public List<Student> getStudentList() {
        return studentList;
    }
    public void setStudentList(List<Student> studentList) {
        this.studentList = studentList;
    }
}
```

Generics AND POLYMORPHISM

When hostellerList is added to studentList in StudentDAO class using addStudentList method, it results in compilation error,

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

public class Main {

        public static void main(String[] args) {

            Hosteller hostellerObj1=new Hosteller(102,"Dravid",
                                                  80000,40000,20000);
            Hosteller hostellerObj2=new Hosteller(103,"Dravid",
                    80000,40000,20000);
            Hosteller hostellerObj3=new Hosteller(104,"Dravid",
                    80000,40000,20000);

            List<Hosteller> hostellerList = new ArrayList<Hosteller>();
            hostellerList.add(hostellerObj1);
            hostellerList.add(hostellerObj2);
            hostellerList.add(hostellerObj3);

            StudentDAO dao = new StudentDAO();
The method addStudentList(List<Student>) in the type StudentDAO is not applicable for the arguments (List<Hosteller>)
            dao.addStudentList(hostellerList);

            System.out.println(dao.getStudentList());
        }
}
```

This is because List<Student> studList= new ArrayList<Hosteller>(); is invalid in generics. Both the types must be the same.

Generics AND POLYMORPHISM

To overcome the error, change the parameter to addStudentList as List<? extends Student> instead of List<Student>

Updated StudentDAO class

```
import java.util.*;

public class StudentDAO {

    List<Student> studentList = new ArrayList<Student>();

    public void addStudentList( List<? extends Student> studentList ){
        this.studentList.addAll(studentList);
    }
    public List<Student> getStudentList() {
        return studentList;
    }
    public void setStudentList(List<Student> studentList) {
        this.studentList = studentList;
    }
}
```

Output :

[ID 102 Name Dravid Tution Fees 80000.0
Hostel Fees 40000.0 Mess Fees 20000.0,
ID 102 Name Dravid Tution Fees 80000.0 Hostel Fees 40000.0 Mess Fees 20000.0, ID 102 Name Dravid Tution Fees 80000.0 Hostel Fees 40000.0 Mess Fees 20000.0]

Collections class

Collections class in java.util package extends directly from Object class

It consists of static methods that operate on or return collection

Few methods in Collections class are

Method	Description
Collections.sort(List<T extends Comparable<? super T>> list)	Sorts the given list in ascending order according to the natural ordering
Collections.binarySearch(List<T extends Comparable<? super T>> list, T key)	Searches the element T in the given list. It returns the index of the searched element if found; else (-) insertion point.
Collections.shuffle(List<?> list)	Randomly shuffles the elements in the list. This will return different results in different calls.
Collections.reverse(List<?> list)	Reverses the order of the elements in the specified list.

Collections example

- In the example, we have used various collections method to do a particular task..
- For the sort method, we have passed the arraylist and the elements will be sorted in the natural order.
- The binarysearch method, takes the arraylist and the elements as argument and it will return the index position of that element.
- The shuffle method, takes the arraylist as argument and shuffle the elements

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class CollectionsDemo {

    public static void main(String[] args) {
        List<String> nameList = new ArrayList<String>();
        nameList.add("Peter");
        nameList.add("Albert");
        nameList.add("Jim");
        nameList.add("Pinky");
        nameList.add("James");
        nameList.add("John");

        // Print elements of nameList
        System.out.println("The collection is "+nameList);

        // Sorting List in ascending order according to the natural ordering
        Collections.sort(nameList);
        System.out.println("Sorted List: "+nameList);

        // Searching a name from list
        int searchIndex = Collections.binarySearch(nameList, "Pinky");
        if(searchIndex >=0)
        {
            System.out.println("Name found at index "+searchIndex);
        }

        else
        {
            System.out.println("Name not found.");
        }

        //Shuffles the list
        Collections.shuffle(nameList);
        System.out.println("Shuffled List: "+nameList);
        }
}
```

Arrays class

Present in java.util package

ARRAYS CLASS HAS A COLLECTION OF STATIC METHODS TO WORK WITH ARRAYS LIKE SORTING AND SEARCHING

FEW METHODS IN ARRAYS CLASS ARE:

- equals
- copyOf
- sort

Arrays - example

Example for equals method

```
import java.util.Arrays;

public class ArraysDemo {

    public static void main(String[] args) {
        int  a[]={2,4,6,8,10};
        int b[]={2,4,6,8,10};
        System.out.println(Arrays.equals(a,b)); //returns  true

        int  c[]={2,4,6,8,10};
        int d[]={10,8,4,2,6};
        System.out.println(Arrays.equals(c,d)); //returns  false


    }

}
```

The equals method accepts two arguments, both the arguments needs to be an array.

Both arrays should be of the same data type and one dimensional

Return true if both arrays contain same elements in the same order

Arrays - example

Example for copyOf method

```
import java.util.Arrays;

public class ArraysDemo {

    public static void main(String[] args) {
        int  num1[]={2,4,6,8,10};
        int num2[]=Arrays.copyOf(num1,3);

        for(int num : num2)
        System.out.println(num);   // prints 2 4 6

        int num3[]={1,2,3};

        int num4[]=Arrays.copyOf(num3,5);
         for(int num : num4)
        System.out.println(num);   // prints  1 2 3 0 0


    }
}
```

The copyOf method accepts two arguments, the first should be an array and the second should be the number of elements that should be copied.

This method returns an array which is a copy of the originalArray.

It copies elements from 0 index position.

If the number of elements exceeds the length of originalArray, it pads those positions with default value.

Arrays - example

Example for sort method

```
import java.util.Arrays;

public class ArraysDemo {

    public static void main(String[] args) {
        int  num1[]={12,41,32,16,10};
        Arrays.sort(num1);

        for(int num : num1)
            System.out.println(num);  // prints 10 12 16 32 41

        int  num2[]={82,41,32,16,10,46,72,89};
        Arrays.sort(num2,1,4); //sorts the numbers from 1  (inclusive) to 4 (exclusive)

        for(int num : num2)
        System.out.println(num);  //prints 82 16 32 41 10 46 72 89


    }
}
```

The sort method sorts the array in ascending order.

We can also sort the elements in a specific range.

Arrays - example

Example to sort user defined object

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

class Employee{
   int id;   String name;
   double salary;
   public Employee(int id, String name, double salary) {
        super();
        this.id = id;
        this.name = name;
        this.salary = salary;
   }
   public String getName()
   {
        return name;
   }
   public String toString()    {
        return "ID : "+id+" Name : "+name+" Salary : "+salary;
   }
}

class EmpNameComp implements Comparator
{
  public int compare(Object obj1, Object obj2) {
      Employee employeeObj1=(Employee)obj1;
      Employee employeeObj2=(Employee)obj2;

      return employeeObj1.getName().compareTo(employeeObj2.getName());
  }
}
```

```
public class ArraysExample {

public static void main(String[] args) {

    List<Employee> empList = new ArrayList<Employee>();
    empList.add(new Employee(107,"John",20000));
    empList.add(new Employee(105,"John",30000));
    empList.add(new Employee(102,"Tom",40000));
    empList.add(new Employee(108,"John",25000));

    Object employeeObj[]=empList.toArray(); //To convert the collection to an Array

    Arrays.sort(employeeObj,new EmpNameComp());

    for(Object obj : employeeObj){
        Employee empObj=(Employee)obj;
        System.out.println(empObj);
    }
  }
}
```

In this example, we have used toArray() function to convert the collection into an array.`

We have passed the array and comparator as an argument to the sort method. The sort method will sort the object based on the logic written in the comparator class. Here we are sorting the objects based on name.

Stream API

TEKNOTURF

Stream represents a sequence of objects from a source, which supports aggregate operations

- The source here refers to a Collection, IO Operation or Arrays that provide data to a Stream
- Stream performs automatic iteration, where as in collection explicit iteration is needed
- Stream keeps the order of the data as it is in the source
- The operations can be executed in series or in parallel
- Like functional programming languages, Streams support Aggregate Operations
- Common aggregate operations are filter, map, reduce, find, match, sort

Creation of stream object

TEKNOTURF

Collection interface has two methods to generate a stream

stream()

- Returns a sequential stream considering collection as its source

parallelStream()

- Returns a parallel Stream considering collection as its source

Methods in Stream API

Various intermediate methods used in stream are:

- distinct
- limit
- sorted
- filter
- min
- max

The terminal operations, is invoked at the end of the pipeline. It will close the stream in some meaningful way. Example : foreach method

distinct - Example

In this example, we have used the asList() method of arrays class to convert the set of values into a collection.

We have created the stream and invoked the distinct() method, this method will return a stream consisting of the distinct elements.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamDemo {
  public static void main(String[] args) {
      List<Integer> numbers = Arrays.asList(3,2,2,3,7,3,5);
      Stream stream = numbers.stream();  //Creating a stream

      Stream stream1 = stream.distinct(); //Fetch the distinct elements and return a new stream
  }
}
```

collect - Example

The collect method is used to get list, map or set from a stream.

The stream will not affect the original collection.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamDemo {

  public static void main(String[] args) {
      List<Integer> numbers = Arrays.asList(3,2,2,3,7,3,5);
      Stream stream = numbers.stream(); //Creating a stream

      Stream stream1 = stream.distinct(); //Fetch the distinct elements
```

```
    List<Integer> distinctList = (List<Integer>)
    stream1.collect(Collectors.toList());

    System.out.println(distinctList);

  }

}
```

Output:
[3, 2, 7, 5]

limit - Example

We have created the stream and invoked the limit() method.
This method will return a stream of first 3 elements.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamDemo {

  public static void main(String[] args) {
      List<Integer> numbers = Arrays.asList(3,2,2,3,7,3,5);
      Stream stream = numbers.stream(); //Creating a stream

      Stream stream1 = stream.limit(3); //Returns a new stream
                                        //of first three elements.
      List<Integer> distinctList = (List<Integer>)
      stream1.collect(Collectors.toList());

      System.out.println(distinctList);

  }

}
```

Output:
[3, 2, 2]

filter - Example

The filter method is used to eliminate elements based on a criteria.

```
import java.util.Arrays;
import java.util.List;

public class StreamDemo {

  public static void main(String[] args) {
            List<String> strings = Arrays.asList("apple","","orange","pineapple","","lime");
            long count=strings.stream().filter(string -> string.isEmpty()).count();
            System.out.println(count);

  }

}
```

Output:
[2]

forEach - Example

An easy way to loop over the stream elements is usage of forEach loop.

```
import java.util.Arrays;
import java.util.List;

public class StreamDemo {

  public static void main(String[] args) {
            List<String> strings = Arrays.asList("apple","","orange","pineapple","","lime");
            strings.stream().filter(string -> !(string.isEmpty())).forEach(System.out::println);
  }
}
```

Output:
apple
orange
pineapple
lime

filter - Example

The 'filter' method is used to eliminate elements based on a criteria

```
import java.util.*;

public class StreamDemo {

  public static void main(String[] args) {
            List<Employee> employees = new ArrayList<Employee>();
            employees.add(new Employee(1,"john",20000));
            employees.add(new Employee(2,"tom",30000));
            employees.add(new Employee(3,"tom",10000));
            employees.stream().filter(e -> e.getSalary() > 15000).forEach(e->System.out.println(e));

  }
}
```

toArray - Example

To convert the stream to an array

```
import java.util.*;

public class StreamDemo {

  public static void main(String[] args) {
            List<Employee> employees = new ArrayList<Employee>();
            employees.add(new Employee(1,"john",20000));
            employees.add(new Employee(2,"tom",30000));
            employees.add(new Employee(3,"tom",10000));
            Employee e[]=(Employee[])employees.stream().toArray();

  }
}
```

Collectors

Collectors are used to summarize the results based on the criteria.

The various aggregate operations performed are:

- Sum
- Average
- Min
- Max

Collectors – Sum Example

To perform the sum operation use Collectors.summingXXX() with Stream.collect()

```
import java.util.*;
import java.util.stream.Collectors;
public class Test {

        public static void main(String ap[])
        {
                List<Employee> employees = new ArrayList<Employee>();
                employees.add(new Employee(1,"john",20000));
                employees.add(new Employee(2,"tom",30000));
                employees.add(new Employee(3,“ram",10000));
                 double total =
employees.stream().collect(Collectors.summingDouble(Employee::getSalary));
            System.out.println("Total Employees Salary : " + total);
        }
}
```

Output:
Total Employees Salary : 60000.0

Collectors – Average Example

To perform the average operation use Collectors. averagingXXX() with Stream.collect()

```
import java.util.*;
import java.util.stream.Collectors;
public class Test {

        public static void main(String ap[])
        {
                List<Employee> employees = new ArrayList<Employee>();
                employees.add(new Employee(1,"john",20000));
                employees.add(new Employee(2,"tom",30000));
                employees.add(new Employee(3,“ram",10000));
                double average =
employees.stream().collect(Collectors.averagingDouble(Employee::getSalary));
                 System.out.println("Avg Salary : " + average);      }
}
```

Output:
Avg Salary : 20000.0

Collectors – max Example

To perform the max operation use Collectors. maxBy() with Stream.collect()

```
import java.util.*;
import java.util.stream.Collectors;
public class Test {

        public static void main(String ap[])
        {
                List<Employee> employees = new ArrayList<Employee>();
                employees.add(new Employee(1,"john",20000));
                employees.add(new Employee(2,"tom",30000));
                employees.add(new Employee(3,“ram",10000));
                System.out.println(“Employee name
:”+employees.stream().collect(Collectors.maxBy(Comparator.comparingDouble(Employee::getSalary))).get().get
EmpName());
}
```

Output:
Employee name : tom

Stream API - Example


```
List<Employee> employees = new ArrayList<Employee>();
                        employees.add(new Employee(1,"john",20000));
                        employees.add(new Employee(2,"tom",30000));
                        employees.add(new Employee(3,"tim",10000));
```

```
List<Employee> list=employees.stream().filter(e->e.getEmpName().startsWith("t")).collect(Collectors.toList());
            for(Employee e:list)
            {
                        System.out.println(e.getEmpName());
            }
```

Stream API - Example


```
List<Employee> employees = new ArrayList<Employee>();
                        employees.add(new Employee(1,"john",20000));
                        employees.add(new Employee(2,"tom",30000));
                        employees.add(new Employee(3,"tim",10000));
```

```
List<Integer> list=employees.stream().map(Employee::getEmpName).map(String::length).collect(Collectors.toList());
                        for(Integer i:list)
                        {
                                    System.out.println(i);
                        }
```