**Module 2**

**The collections and Framework: Collections Overview, Recent Changes to Collections, The Collection Interfaces, The Collection Classes, Accessing a collection Via an Iterator,Storing User Defined Classes in Collections, The Random Access Interface, Working With Maps, Comparators, The Collection Algorithms, Why Generic Collections?, The legacy Classes and Interfaces, Parting Thoughts on Collections.**

**Introduction to Generics : Generic class with one type/generic parameter (RAW TYPE)**

```
class swap<T> { // T is a TYPE PARAMETER
   private T i,j;
   public void assign(T a, T b)  {
      i=a; j=b;
      System.out.println(i.getClass());
   }

   public void swap()   {
      T t=i;
      System.out.println(t.getClass());
      i=j;    //Assignment or binary bits copy can be done irrespective of the type
      j=t;
   }
   public void display()
   {  System.out.println(i+" "+j);   }
```

**Introduction to Generics : Generic class with one type/generic parameter (RAW TYPE)**

```
public T acess()
    {  return i; }
    public void modify(T a)
    {i=a;}
}
public class Test
{
        public static void main(String[] args) {
        swap<Integer> a = new swap<Integer>();
// Integer is a TYPE ARGUMENT
        a.assign(1,2);
        a.display();
        Integer t = a.acess();          t=t+10;        a.modify(t);
         a.display();
        a.swap();    a.display();
        }      }
```

**Introduction to Generics : Generic class with more than one type/generic parameter (RAW TYPE)**

```
import java.util.*;

class generic<u, v> {
        u a;
        v b;
        public void assign(u p, v q)
        {
                a=p; b=q;
        }
        public void display()
        {
                System.out.println(a+ " "+b);
        }
}
```

**Introduction to Generics : Generic class with more than one type/generic parameter (RAW TYPE)**

```
public class Test  {
    public static void main(String[] args) {
        generic<Integer,Float> a = new generic<Integer,Float>();
        a.assign(1, (float)1.2);
        a.display();

        generic<String,Integer> b = new generic<String,Integer>();
        b.assign("tesla",10);
        b.display();

    } }
```

**Introduction to Generics - Generic methods - class level**

Like generic classes, generic methods can be coded.

```
public class Test  {
        public static <T> void disp(T a)
        { System.out.println(a); }

        public static void main(String[] args) {
        Integer a = 10;          disp(a);
        String p="aaa";          disp(p);          }  }
```

For static methods <T> is always placed before the return type of the method.

It indicates that the T identifier is a type parameter, to distinguish it with concrete types.

If the type parameter of a non-static/instance-level generic method is same as the enclosing class, the indicator <T> is not required.

## Introduction to Generics - Generic methods - Instance level

```
class disp<T>
{
        public void disp1(T a)
        {  System.out.println(a);  }
}
public class Test  {
        public static void main(String[] args) {
        disp<Object> r = new disp<Object>();
        Integer a = 10;

        r.disp1(a);

        String p="aaa";
        r.disp1(p);
}  }
```

**disp is a generic class and it's parameter T is also generic.**

**Introduction to Generics - Generic types**

Generic type in java can be declared using two ways

1. Raw generic type
2. Bounded generic type

**Bounded generic type**

The types that are used as **type arguments** can be restricted.
Ex: a method that operates on numbers might only want to accept instances of **Number** or its subclasses.

This is where *bounded type parameters* will be useful.

To declare a bounded type parameter, list the type parameter name, followed by the extends keyword, followed by its *upper bound*, which in this example is **Number**.

In this context, **extends** is used in a general sense to mean either "**extends**" (as in classes) or "**implements**" (as in interfaces).

*****

Generic methods allow type parameters to be used to express dependencies among the types of one or more arguments to a method and/or its return type. If there isn't such a dependency, a generic method should not be used.***** (Applies for Raw and Bounded generic type)

**Introduction to Generics - Bounded type parameters in class**

The type of parameter to a generic type can be restricted to be a subtype of concrete type.

```
interface base1
{
        void assign(int a, int b);
        void swaping();
        void display();
}
class swap1 implements base1 {
   private int i,j;
   @Override
   public void assign(int a,int b) { i=a; j=b;   }
   @Override
   public void swaping()  {  int t=i;  i=j; j=t;  }
   @Override
   public void display() { System.out.println(i+" "+j); }
}
```

**Introduction to Generics - Bounded type parameters in class**

```
class generic<T extends base1>  {
        T a;
        public void disp(T p)
        {   a=p;
            a.assign(1, 2);        a.display();        a.swaping();        a.display();
//above statements are approved because in bounded type, the type of base1 class is specific and the member
//function belongs to base1, which are inherited and overridden.
        }  }
class Test  {
        public static void main(String[] args) {
        swap1 c= new swap1();

        generic<swap1> b = new generic<swap1>();
        b.disp(c);

        generic<Integer> c=new generic<Integer>();  //CTE
        }  }
```

CTE because built-in class Integer is not derived class of base1.

**Introduction to Generics - Bounded type parameters in Generic methods - Manipulation on data members**

```
import java.util.*;
interface add_interface
{
        public void add( Object q);
}

class add_int implements add_interface
{
        Integer a;
        public add_int(int r) { a=r; }

        @Override
        public void add( Object q) {
           add_int b=(add_int)q;
          System.out.println("int "+(a+b.a));
        }
}
```

**Introduction to Generics - Bounded type parameters in Generic methods - Manipulation on data members**

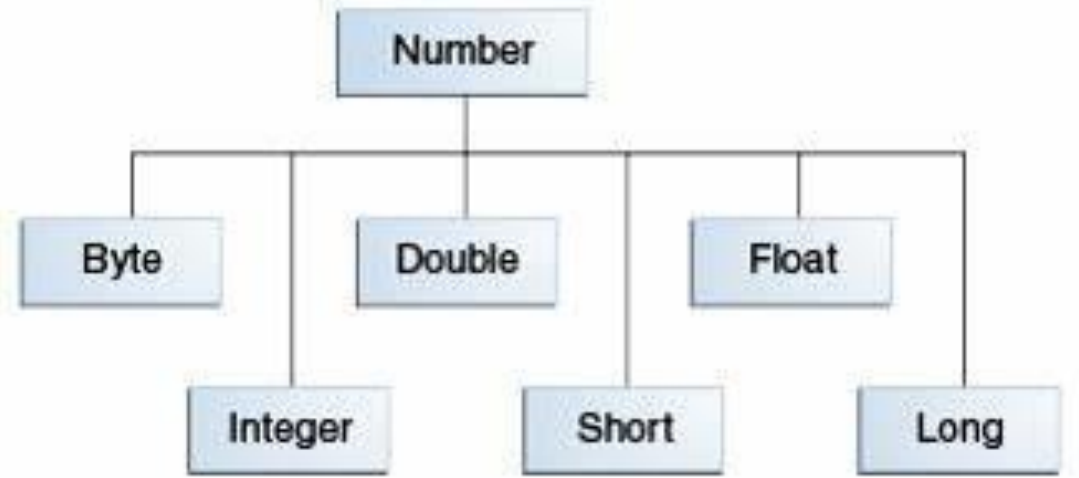
```
class add_string implements add_interface   {
        String a;
        public add_string(String s) { a=s; }

        @Override
        public void add(Object q) {
                add_string b=(add_string)q;
                System.out.println("String "+(a+b.a));  }   }

class adding1<T extends add_interface>          {
        public void add(T p, T q,String s) // generic method bounded type
        {
                System.out.println(s);
                p.add(q);
//add( ) invocation is approved because base class has the method and it's name has been used in bounded type
        } }
```

**Introduction to Generics - Bounded type parameters in Generic methods - Manipulation on data members**

```
public class Test  {
    public static<T extends add_interface> void add(T p, T q, String s)
    {  //static method bounded type
        System.out.println(s);
        p.add(q);   }
    public static void main(String[] args) {
        add_int a = new add_int(1);                    add_int b = new add_int(1);

        Test.add(a,b,"Calling Static function(for int add)");

        adding1<add_int> c = new adding1<add_int>();
        c.add(a,b,"Calling Member function(for int add)");

        add_string p = new add_string("rnsit");
        add_string q = new add_string("jssate");
        Test.add(p, q,"Calling Static function(for string add)" );

        adding1<add_string> d = new adding1<add_string>();
        d.add(p, q, "Calling Member function(for string add");                } }
```

**Introduction to Generics - Number class in java**

Number is an interface in java, which is implemented by Integer, Float, Double ….




Hence, an instance of derived class types can be pointed by the reference of type Number.

Ex: Integer c=1;

Number b=c; System.out.println(b); Double d=10.2;

b=d; System.out.println(b);

**Introduction to Generics - Number class in java**

Similarly, an ArrayList of Number type collection can hold on to any type of numeric values.

Collection framework is used to store instances or objects of same type.

If the specific type of collection is Number, then different types of numeric values can be kept in that collection.

```
import java.util.*;

public class Test  {
        public static void main(String[] args) {
        ArrayList<Number> a = new ArrayList<Number>();
        a.add(100);
        a.add(100.2);
        a.add((float)100.2);
        a.add((long)300);

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

**Introduction to Generics - Wildcard**

In generic code, the question mark (?), called the *wildcard*, represents an unknown type.

The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific).

The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

Wildcard '?' are used in 3 different ways in java

1. Upper bounded wildcard
2. Lower bounded wildcard
3. Unbounded wildcard

## Introduction to Generics - Wildcard

### Upper bounded wildcard

Upper bounds in wild cards is similar to the bounded type in generics.

Using this all the **subclasses** of a particular **superclass** can be used as a typed parameter.
Keyword used is “extends”

### Lower bounded wildcard

Using this all the **superclasses** of a particular **subclass** can be used as a typed parameter.
Keyword used is “super”.

### Unbounded wildcard

An unbounded wildcard is the one which enables the usage of all the subtypes of Object, which is accepted as typed-parameter.

**Introduction to Generics - Wildcard**

```
import java.util.*;
public class Test {
    public static void disp(Collection<? extends Number> a){  //upper bound wildcard
        Iterator x = a.iterator();
        for(;x.hasNext();)
            System.out.println(x.next());
    }

    public static void disp1(Collection<? super Integer> a){ //lower bound wildcard
        Iterator x = a.iterator();
        for(;x.hasNext();)
            System.out.println(x.next());
    }

    public static void disp2(Collection<?> a){ // unbounded wildcard
        Iterator x = a.iterator();
        for(;x.hasNext();)
            System.out.println(x.next());    }   }
```

## Introduction to Generics - Wildcard on built-in collections/generic types

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

    ArrayList<Integer> a = new ArrayList<Integer>();
    a.add(1); a.add(2);  a.add(3);
    System.out.println("Output from upper bounded method");
    disp(a);
    System.out.println("Output from lower bounded method");
    disp1(a);
    System.out.println("Output from unbounded method");
    disp2(a);
```

## Introduction to Generics - Wildcard on built-in collections/generic types

```
ArrayList<Double> b = new ArrayList<Double>();
b.add(1.1); b.add(1.2);
b.add(3.2);
System.out.println("Output from upper bounded method");
disp(b);
System.out.println("Output from lower bounded method");
disp1(b); //CTE
System.out.println("Output from unbounded method");
disp2(b);
```

## Introduction to Generics - Wildcard on built-in collections/generic types

```
        ArrayList<Object> c=new ArrayList<Object>();
        c.add(10.2);
        c.add(1);
        c.add("rnsit");
        System.out.println("Output from lower bounded method");
        //disp(c);  //CTE
        System.out.println("Output from upper bounded method");
        disp1(c);
        System.out.println("Output from unbounded method");
        disp2(c);


    }
}
```

## Introduction to Generics - Observation on generic class

If a generic class is being instantiated and no type argument is passed to it, then the compiler will automatically consider the type argument as "Object".

**Ex:**

```
class generic<T>
{
    public void disp(T b)
    {
        System.out.println(b);
        System.out.println(b.getClass());
    }
}
```

**Introduction to Generics - Observation on generic class**

```
class cmp   {
        @Override
        public String toString()
        {
                return "in cmp";
        }
}
class Test {
        public static void main(String[] args) {
                generic a = new generic();   //without type argument, default is Object
                Integer b = 10;       a.disp(b);

                cmp c = new cmp();     a.disp(c);
        }   }
```

**Introduction to Generics - Wildcard on user-defined generic types**

```
import java.util.*;
class tst<T>
{
        protected T a;
        void disp() {}
}

class c1 extends tst  {  //generic interface to specific class inheritance
                                //T will be type of Object.
        public void disp()   {
                a = new Object();
                System.out.println("in class c1");
                System.out.println(a.getClass());
        }
}
```

Reference of a class derived from Object cannot be used to point to an instance of type Object.
Hence, reference ‘a’ is of type generic.

## Introduction to Generics - Wildcard on user-defined generic types

```
public class Test {
    public static void disp( tst<? extends tst> a)  //upper bound
    {
        a.disp();
    }

    public static void disp1(tst<? super tst> a) //lower bound
    {
        a.disp();
    }

    public static void disp2(tst<?> a) //unbound
    {
        a.disp();
    }
```

**Introduction to Generics - Wildcard on user-defined generic types**

```
public static void main(String[] args)
        {
                c1 a= new c1();
                disp(a);

                disp1(a);

                disp2(a);

        }
}
```

**Introduction to Generics - Usage of wild card (?)**

**Ex:**

```
class generic<T> { }
    class te {
        public void add(generic<?> g)  {}
    }
    class Test {
        public static void main(String[] args) {
        te a = new te();
        a.add(new generic<String>());
        }
    }
```

## Collections

"java.util" is an important package which contains a large number of classes and interfaces that support a broad range of functionality.

Ex: java.util has classes that generate pseudorandom numbers, manage date and time, observe events, manipulate sets of bits, tokenize strings, and handle formatted data.

java.util package also contains one of Java's most powerful subsystems:
The *Collections Framework*

**The Collections Framework is a sophisticated hierarchy of interfaces and classes for managing groups of objects.**

**Collections**

Interfaces defined by java.util package are

| | | |
|---|---|---|
| Collection | List | Queue |
| Comparator | ListIterator | RandomAccess |
| Deque (Added by Java SE 6.) | Map | Set |
| Enumeration | Map.Entry | SortedMap |
| EventListener | NavigableMap (Added by Java SE 6.) | SortedSet |
| Formattable | NavigableSet (Added by Java SE 6.) | |
| Iterator | Observer | |

## Collections

**java.util** list of **classes** are as follows.

| | | |
|---|---|---|
| AbstractCollection | EventObject | Random |
| AbstractList | FormattableFlags | ResourceBundle |
| AbstractMap | Formatter | Scanner |
| AbstractQueue | GregorianCalendar | ServiceLoader (Added by Java SE 6.) |
| AbstractSequentialList | HashMap | SimpleTimeZone |
| AbstractSet | HashSet | Stack |
| ArrayDeque (Added by Java SE 6.) | Hashtable | StringTokenizer |
| ArrayList | IdentityHashMap | Timer |
| Arrays | LinkedHashMap | TimerTask |
| BitSet | LinkedHashSet | TimeZone |
| Calendar | LinkedList | TreeMap |
| Collections | ListResourceBundle | TreeSet |
| Currency | Locale | UUID |
| Date | Observable | Vector |
| Dictionary | PriorityQueue | WeakHashMap |
| EnumMap | Properties | |
| EnumSet | PropertyPermission | |
| EventListenerProxy | PropertyResourceBundle | |

## Collections - Overview

. Specify the usage of Collection framework. Explain the facility that is rendered by the same

Java Collections Framework standardizes the way in which groups of objects are handled by programs.

Entire Collections Framework is built upon a set of standard interfaces.

Another entity closely associated with the Collections Framework is the **Iterator** interface.

An *iterator* offers a general-purpose, standardized way of accessing the elements within a collection, one at a time.

Each collection implements **Iterator**, the elements of any collection class can be accessed through the methods defined by **Iterator**.

In addition to collections, the framework defines several map interfaces and classes.

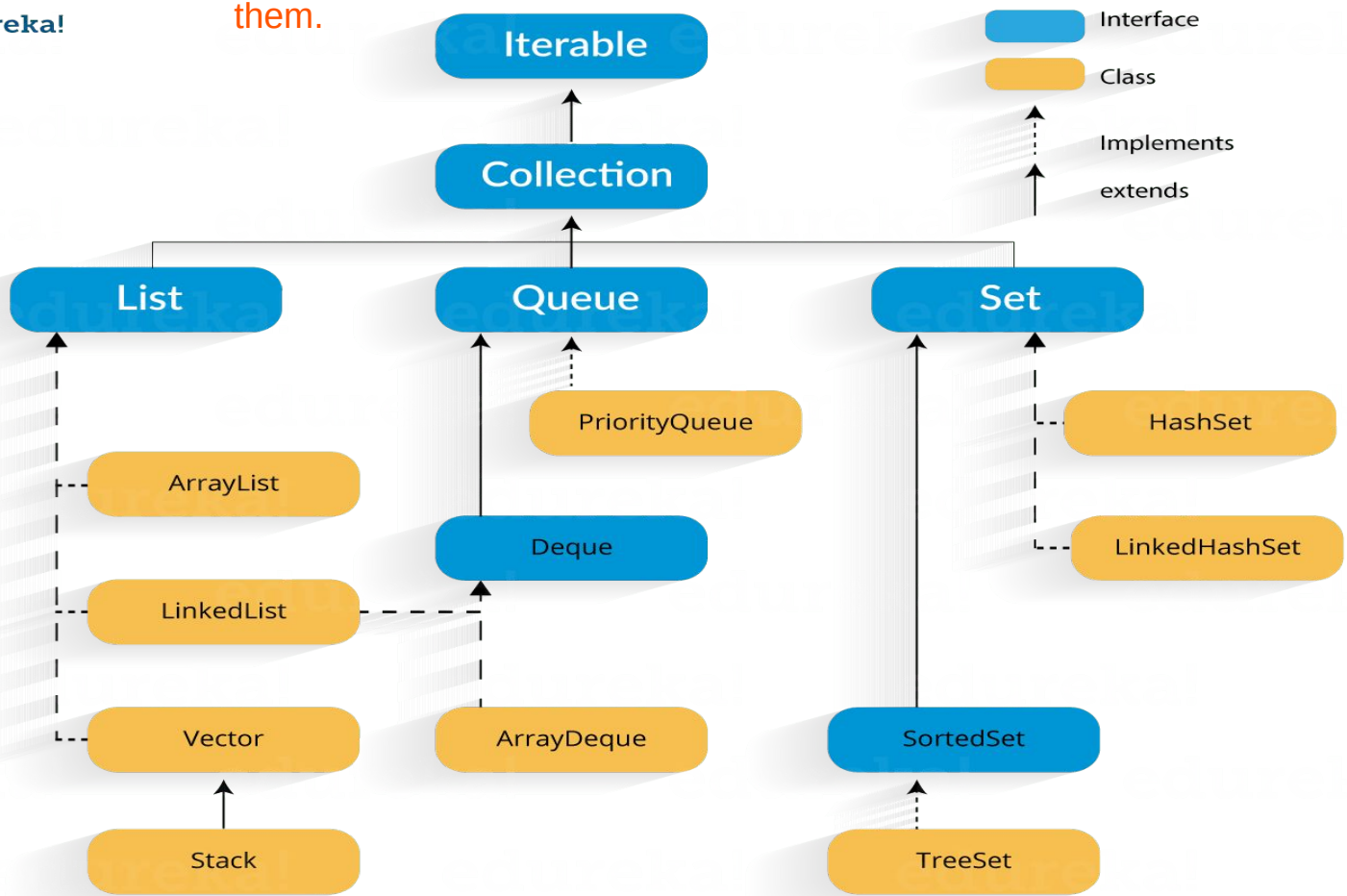
*Maps* store key/value pairs.

Contents of a map can be processed as a collection. Collections provide a better way of doing several things.

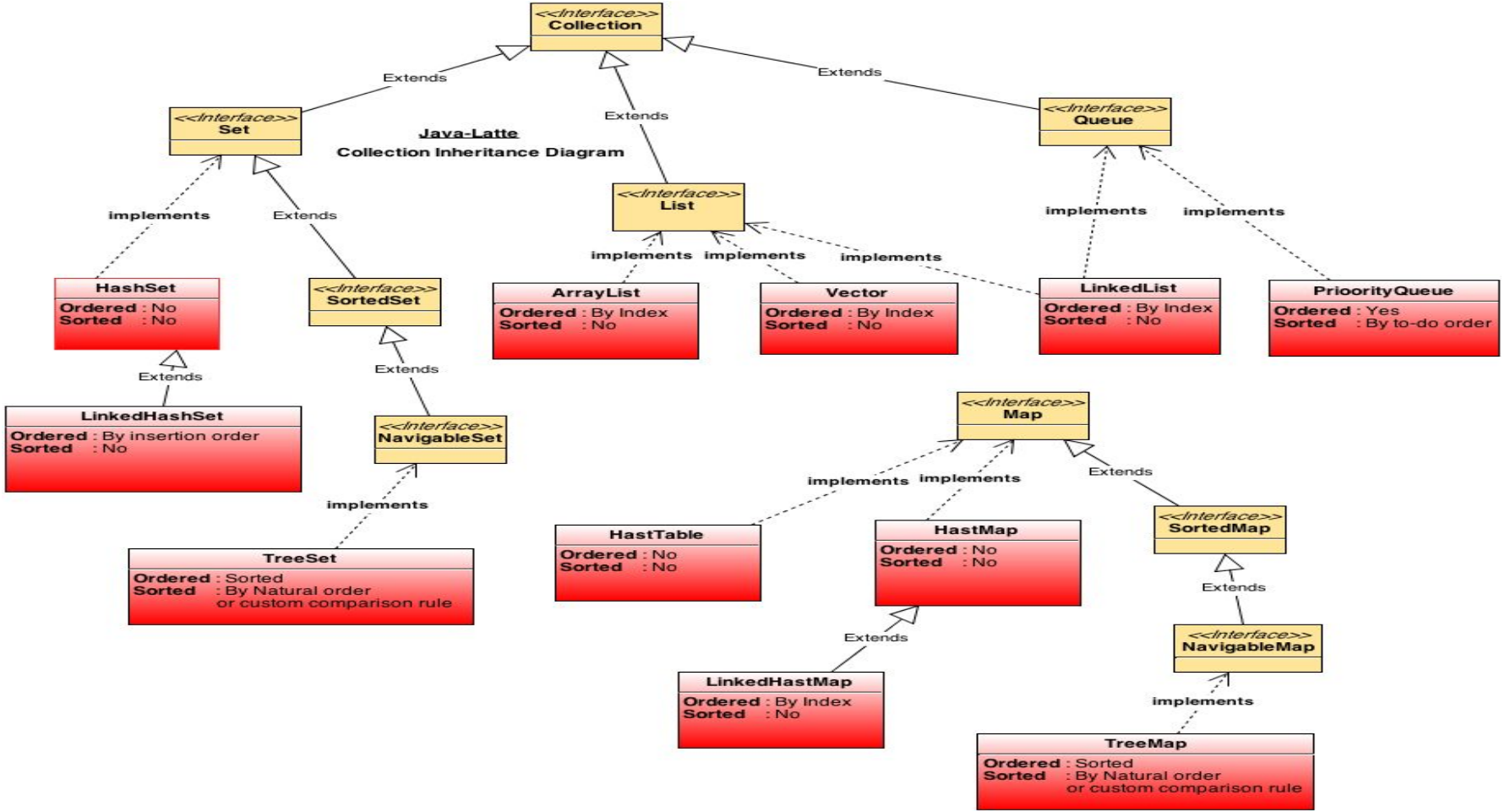
# Collections - Overview

With an hierarchical diagram of Collections, identify the interface and concrete classes in them.

edureka!



# Collections - Overview



## Collections - Recent Changes to Collections

Explain the recent changes made to Collection framework, which makes it more serene to use

The changes in Collection framework are caused by the addition of generics, autoboxing/unboxing, and the for-each style.

### 1. Generics Fundamentally Change the Collections Framework

All collections are now generic, and many of the methods that operate on collections take generic type parameters.

Generics add the one feature that collections had been missing: **type safety.**

(Prior to generics, all collections stored **Object** references, which meant that any collection could store any type of object. Thus, it was possible to accidentally store incompatible types in a collection. Doing so could result in run-time type mismatch errors.)

With generics, it is possible to explicitly state the type of data being stored, and run-time type mismatch errors can be avoided.

## Collections - Recent Changes to Collections

### 2. Autoboxing Facilitates the Use of Primitive Types

Autoboxing/unboxing facilitates the storing of primitive types in collections.

**A collection can store only references, not primitive values.**

Because of autoboxing/unboxing, Java can automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types. There is no need to manually perform these operations.

### 3. The For-Each Style for Loop

All collection classes in the Collections Framework have been retrofitted to implement the **Iterable** interface, which means that a collection can be cycled through by use of the for-each style **for** loop.

**Collections - The Collection Interfaces**

The Collections Framework defines several interfaces.

List out the important interfaces defined in Collection framework. Differentiate between modifiable and unmodifiable collections

**Collection interfaces** determine the fundamental nature of the collection classes.

Concrete classes provide different implementations of the standard interfaces.

Interfaces that support collections are summarized below.

| Interface | Description |
|---|---|
| Collection | Enables you to work with groups of objects; it is at the top of the collections hierarchy. |
| Deque | Extends **Queue** to handle a double-ended queue. (Added by Java SE 6.) |
| List | Extends **Collection** to handle sequences (lists of objects). |
| NavigableSet | Extends **SortedSet** to handle retrieval of elements based on closest-match searches. (Added by Java SE 6.) |
| Queue | Extends **Collection** to handle special types of lists in which elements are removed only from the head. |
| Set | Extends **Collection** to handle sets, which must contain unique elements. |
| SortedSet | Extends **Set** to handle sorted sets. |

## Collections - The Collection Interfaces

In addition to the collection interfaces, collections also use the **Comparator**, **RandomAccess**, **Iterator**, and **ListIterator** interfaces.

**Comparator** defines how two objects are compared.

**Iterator** and **ListIterator** enumerate the objects within a collection.

**RandomAccess** supports efficient, random access to elements.

**Modifiable and unmodifiable collections**
To provide flexibility in their use, the collection interfaces allow some methods to be optional.

The optional methods enable modification of the contents of a collection.

Collections that support these methods are called *modifiable.*

Collections that do not allow their contents to be changed are called *unmodifiable.*

If an attempt is made to use one of these methods on an unmodifiable collection, an **UnsupportedOperationException** is thrown. All the built-in collections are modifiable.

## Collections - The Collection Interfaces

The **Collection** interface is the foundation upon which the Collections Framework is built because it must be implemented by any class that defines a collection.

**Collection** is a generic interface that has this declaration:

**interface Collection<E>**

Here, **E** specifies the type of objects that the collection will hold.

**Collection** extends the **Iterable** interface. This means that all collections can be cycled through by use of the for-each style **for** loop.

**Collection** declares the core methods that all collections will have.

These methods are summarized in Table.

5. Specify the prototypes of add( ), addAll( ), remove( ) and removeALL( ) methods of Collection interface and explain the usage of the same

## Collections - The Collection Interfaces

| Method | Description |
|---|---|
| boolean add(E *obj*) | Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates. |
| boolean addAll(Collection<? extends E> *c*) | Adds all the elements of *c* to the invoking collection. Returns **true** if the operation succeeded (i.e., the elements were added). Otherwise, returns **false**. |
| void clear( ) | Removes all elements from the invoking collection. |
| boolean contains(Object *obj*) | Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**. |
| boolean containsAll(Collection<?> *c*) | Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**. |
| boolean equals(Object *obj*) | Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**. |
| int hashCode( ) | Returns the hash code for the invoking collection. |
| boolean isEmpty( ) | Returns **true** if the invoking collection is empty. Otherwise, returns **false**. |
| Iterator<E> iterator( ) | Returns an iterator for the invoking collection. |

## Collections - The Collection Interfaces

| | |
|---|---|
| boolean remove(Object *obj*) | Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**. |
| boolean removeAll(Collection<?> *c*) | Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**. |
| boolean retainAll(Collection<?> *c*) | Removes all elements from the invoking collection except those in *c*. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**. |
| int size( ) | Returns the number of elements held in the invoking collection. |
| Object[ ] toArray( ) | Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements. |
| <T> T[ ] toArray(T *array*[ ]) | Returns an array that contains the elements of the invoking collection. The array elements are copies of the collection elements. If the size of *array* equals the number of elements, these are returned in *array*. If the size of *array* is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of *array* is greater than the number of elements, the array element following the last collection element is set to **null**. An **ArrayStoreException** is thrown if any collection element has a type that is not a subtype of *array*. |

**TABLE 17-1** The Methods Defined by **Collection**

**Collections - The Collection Interfaces**

```
import java.util.*;
class student {
        private String name, usn;

        public student(String n, String u) {  name=n;  usn=u;  }
        @Override
        public String toString() {  System.out.println("toString called "); return(name+" "+usn);  }
        @Override
        public boolean equals(Object obj)
        {
                System.out.println("equlas method called");
                student b=(student)obj;
                if (name==b.name && usn == b.usn)
                        return true;
                return false;
        }
}
```

**Collections - The Collection Interfaces**

```
class Test {
 public static void main(String[] args){
    ArrayList<student> a = new ArrayList<student>();
    ArrayList<student> b = new ArrayList<student>();

    a.add(new student("1","1"));  a.add(new student("2","2"));
    b.add(new student("3","3"));  b.add(new student("4","4"));

   a.addAll(b);
    System.out.println(a.contains(new student("4","4")) );
    System.out.println(a);
    a.add(new student("1","1"));        a.add(new student("2","2"));
    b.add(new student("2","2"));        b.add(new student("1","1"));
// if order of info is maintained in b (i.e 1 after 2) a.equals returns true
     System.out.println(a.equals(b) );
 }  }
```

6. Explain with a suitable code snippet how ClassCastException and NullPointerException are generated in the Collections framework.

## Collections - The Collection Interfaces

Several of these methods can throw an **UnsupportedOperationException**, this occurs if a collection cannot be modified.

A **ClassCastException** is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a collection.

A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the collection.

An **IllegalArgumentException** is thrown if an invalid argument is used.

An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length collection that is full.

Objects are added to a collection by calling **add( )**, which takes an argument of type **E**, which means that objects added to a collection must be compatible with the type of data expected by the collection.

All the contents of one collection can be added to another by calling **addAll( )**.

An object can be removed by using **remove( )**.

**Collections - The Collection Interfaces - UnsupportedOperationException**

```
import java.util.Arrays;
import java.util.List;
public class test {
   public static void main(String[] args){
       Integer a[] = { 1, 2 };

      List<Integer> l = Arrays.asList(a);
      System.out.println(l);

      l.add(3); // generates java.lang.UnsupportedOperationException
   } }
```

**Arrays** class provides static methods to dynamically create and access **Java arrays**.

It consists of only static methods and the methods of Object class.

The methods of this class can be used by the class name itself.

**Arrays.asList(a)**, returns a fixed-size list of the passed array.

No additional data can be kept in fixed-size List. Hence, l.add(3) generates UnsupportedOperationException.

**Collections - The Collection Interfaces - ClassCastException**

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

        Object obj = new Integer(100);
       System.out.println((String) obj);       }  }  SOP line generates ClassCastException
```

ClassCastException can be prevented using Generics, because Generics provide compile time checks and can be used to develop type-safe applications.

```
public class Main {
   public static void main(String[] args) {
      ArrayList<Integer> i = new ArrayList<Integer>();

      i.add(1);  i.add("2");       }       }
```

i.add(“2”) generates CTE

**Collections - The Collection Interfaces - NullPointerException**

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        TreeSet<Integer> i = new TreeSet<Integer>();
        Integer a=null;
        i.add(1);  i.add(a);

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

The **TreeSet** object doesn't allows null values but, If you try to add them, a runtime exception will be generated.

**Collections - The Collection Interfaces**

To remove a group of objects, **removeAll( )** must be called.

All elements can be removed except those of a specified group by calling **retainAll( )**.

To empty a collection, call **clear( )**.

A specific object can be found by calling **contains( )**.

To determine whether one collection contains all the members of another, call **containsAll( )**.

Whether a collection is empty or not is found by calling **isEmpty( )**.

The number of elements currently held in a collection can be determined by calling **size( )**.

The **toArray( )** methods return an array that contains the elements stored in the invoking collection. The first returns an array of **Object**. The second returns an array of elements that have the same type as the array specified as a parameter.

The second form is more suitable because it returns the desired array type.

**Collections - The Collection Interfaces**

Often, processing the contents of a collection by using array-like syntax is advantageous. By providing a pathway between collections and arrays, the best of both worlds can be used.

Two collections can be compared for equality by calling **equals( )**. The precise meaning of "equality" may differ from collection to collection.

Ex: **equals( )** can be implemented so that it compares the values of elements stored in the collection. Alternatively, **equals( )** can compare references to those elements.

**iterator( )** is a method, which returns an iterator to a collection. Iterators are frequently used when working with collections.

**Collections - The Collection Classes**

Some of the classes provide full implementations of collection interface.

Others are abstract, providing skeletal implementations that are used as starting points for creating concrete collections.

None of the collection classes are synchronized, but it is possible to obtain synchronized versions.

The standard collection classes are summarized in the following table:

## Collections - The Collection Classes

| Class | Description |
|---|---|
| AbstractCollection | Implements most of the **Collection** interface. |
| AbstractList | Extends **AbstractCollection** and implements most of the **List** interface. |
| AbstractQueue | Extends **AbstractCollection** and implements parts of the **Queue** interface. |
| AbstractSequentialList | Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements. |
| LinkedList | Implements a linked list by extending **AbstractSequentialList**. |
| ArrayList | Implements a dynamic array by extending **AbstractList**. |
| ArrayDeque | Implements a dynamic double-ended queue by extending **AbstractCollection** and implementing the **Deque** interface. (Added by Java SE 6.) |
| AbstractSet | Extends **AbstractCollection** and implements most of the **Set** interface. |
| EnumSet | Extends **AbstractSet** for use with **enum** elements. |
| HashSet | Extends **AbstractSet** for use with a hash table. |
| LinkedHashSet | Extends **HashSet** to allow insertion-order iterations. |
| PriorityQueue | Extends **AbstractQueue** to support a priority-based queue. |
| TreeSet | Implements a set stored in a tree. Extends **AbstractSet**. |

**Collections - The ArrayList class**

The **ArrayList** class extends **AbstractList** and implements the **List** interface. ArrayList is a generic class that has this declaration:

class ArrayList<E>

Here, E specifies the type of objects that the list will hold.

ArrayList supports dynamic arrays that can grow as needed.

In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that it must be known in advance how many elements an array will hold.

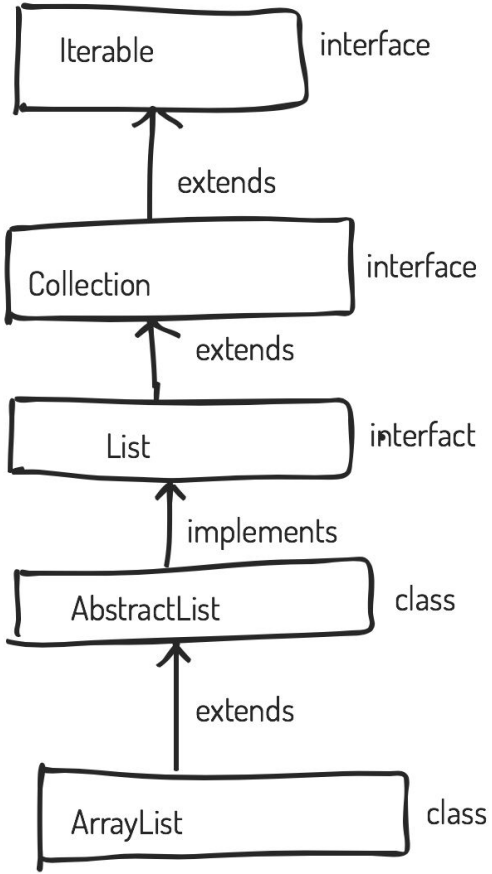
An **ArrayList** is a variable-length array of object references. That is, an ArrayList can dynamically increase or decrease in size.

Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.

**ArrayList** has the constructors shown here:

ArrayList( )
ArrayList(Collection<? extends E> *c*)
ArrayList(int *capacity*)

# Collections - The ArrayList class



## Collections - The ArrayList class

The first constructor builds an empty array list.

The second constructor builds an array list that is initialized with the elements of the collection *c*.

The third constructor builds an array list that has the specified initial *capacity*.

The capacity is the size of the underlying array that is used to store the elements.

The capacity grows automatically as elements are added to an arraylist.

The following program displays use of **ArrayList**. An array list is created for objects of type **String**, and then several strings are added to it. (Consider that a quoted string is translated into a **String** object.) The list is then displayed. Some of the elements are removed and the list is displayed again.

**Collections - The ArrayList class**

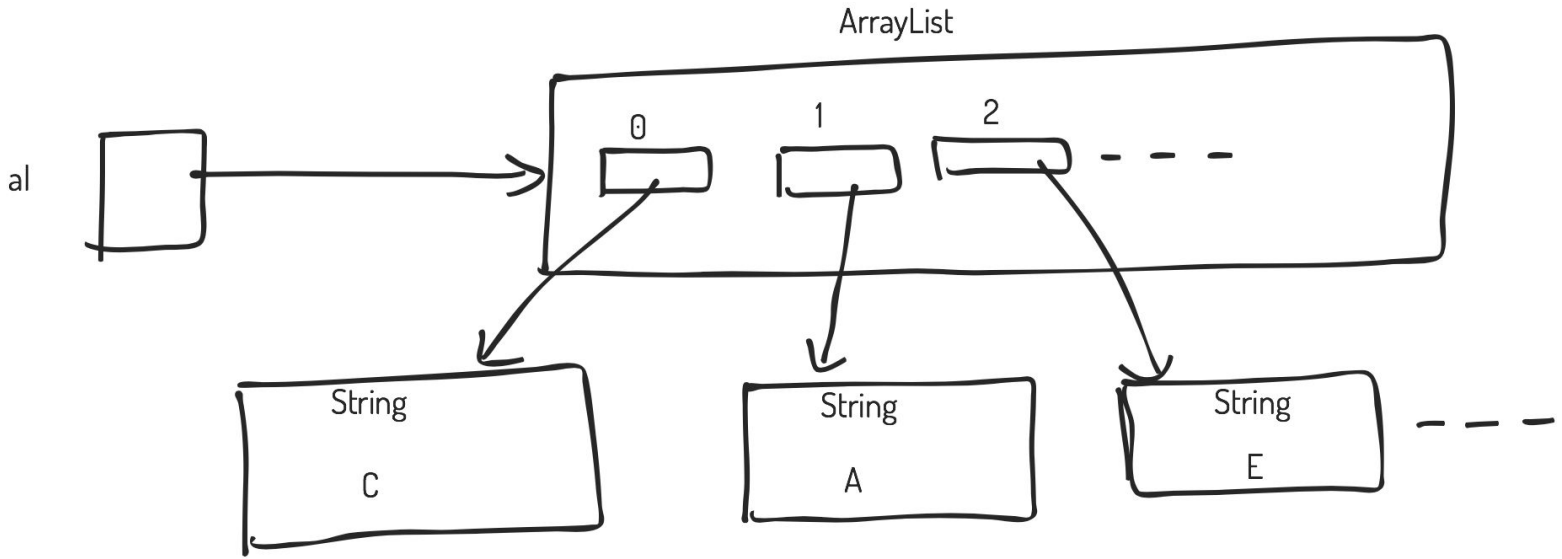
Ex:

```
// Demonstrate ArrayList.
import java.util.*;
class Main {
  public static void main(String args[]) {
    // Create an array list.
    ArrayList<String> al = new ArrayList<String>();
    System.out.println("Initial size of al: " + al.size());

     // Add elements to the array list.
    al.add("C");      al.add("A");      al.add("E");
    al.add("B");      al.add("D");       al.add("F");     al.add(1, "A2");
    System.out.println("Size of al after additions: " + al.size());

    // Display the array list.
    System.out.println("Contents of al: " + al);

    // Remove elements from the array list.
    al.remove("F");           al.remove(2);
    System.out.println("Size of al after deletions: " + al.size());
    System.out.println("Contents of al: " + al);     }        }
```

**Collections - The ArrayList class**

Printing the contents of ArrayList using Iterator

```
Iterator<String> s = al.iterator();

for(;s.hasNext()==true;)
    System.out.println(s.next());
```



**Collections - Obtaining an Array from an ArrayList**

Actual array that contains the contents of the list can be extracted from **ArrayList**, by calling **toArray( )** method, which is defined by Collection.

There are two versions of **toArray( )**,

```
Object[ ] toArray( )
<T> T[ ] toArray(T array[ ])
```

The first returns an array of **Object.**

The second returns an array of elements that have the same type as T.

Second form is more convenient because it returns the proper type of array.

**Collections - Obtaining an Array from an ArrayList**

Ex:

```
// Convert an ArrayList into an array.
import java.util.*;
class Main {
public static void main(String args[]) {
        // Create an array list.
        ArrayList<Integer> al = new ArrayList<Integer>();
        // Add elements to the array list.
        al.add(1);   al.add(2);   al.add(3);          al.add(4);
        System.out.println("Contents of al: " + al);
        // Get the array.
        Integer ia[] = new Integer[al.size()];
        ia = al.toArray(ia);
        int sum = 0;
        // Sum the array.
        for(int i : ia) sum += i;
        System.out.println("Sum is: " + sum);  }  }
```

The program begins by creating a collection of integers. Next, **toArray( )** is called and it obtains an array of Integers. Then, the contents of that array are summed by use of a for-each style for loop.

**Collections - Obtaining an Array from an ArrayList**

Collections can store only references to, not values of, primitive types.

However, autoboxing makes it possible to pass values of type **int** to **add( )** without having to manually wrap them within an Integer.

Autoboxing causes them to be automatically wrapped. In this way, autoboxing significantly improves the ease with which collections can be used to store primitive values.

## Collections - The List Interface

The **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.

Elements can be inserted or accessed by their position in the list, using a zero-based index.

A list may contain duplicate elements.

**List** is a generic interface that has this declaration:

**interface List<E>**

Here, **E** specifies the type of objects that the list will hold.

In addition to the methods defined by **Collection**, **List** defines some of its own, which are summarized in the table below.

Several of these methods will throw an **UnsupportedOperationException** if the list cannot be modified, and a **ClassCastException** is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a list.

# Collections - The List Interface

| Method | Description |
|---|---|
| void add(int *index*, E *obj*) | Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. |
| boolean addAll(int *index*, Collection<? extends E> *c*) | Inserts all elements of *c* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise. |
| E get(int *index*) | Returns the object stored at the specified index within the invoking collection. |
| int indexOf(Object *obj*) | Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned. |

## Collections - The List Interface

| | |
|---|---|
| int lastIndexOf(Object *obj*) | Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned. |
| ListIterator<E> listIterator( ) | Returns an iterator to the start of the invoking list. |
| ListIterator<E> listIterator(int *index*) | Returns an iterator to the invoking list that begins at the specified index. |
| E remove(int *index*) | Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one. |
| E set(int *index*, E *obj*) | Assigns *obj* to the location specified by *index* within the invoking list. |
| List<E> subList(int *start*, int *end*) | Returns a list that includes elements from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object. |

**TABLE 17-2** The Methods Defined by **List**

**Collections - The List Interface**

Some of the methods will throw an **IndexOutOfBoundsException** if an invalid index is used.

A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the list.

An **IllegalArgumentException** is thrown if an invalid argument is used.

To the versions of **add( )** and **addAll( )** defined by **Collection**, **List** adds the methods **add(int, E)** and **addAll(int, Collection)**. These methods insert elements at the specified index.

The semantics of **add(E)** and **addAll(Collection)** defined by **Collection** are changed by **List** so that they add elements to the end of the list.

To obtain the object stored at a specific location, call **get( )** with the index of the object.

To assign a value to an element in the list, call **set( )**, specifying the index of the object to be changed.

To find the index of an object, use **indexOf( )** or **lastIndexOf( )**.

A sublist can be obtained from a list by calling **subList( )**, specifying the beginning and ending indexes of the sublist.

**Collections - The Queue Interface**

The Queue interface extends Collection and declares the behavior of a queue, which is often a first-in, first-out list. Queue is a generic interface that has this declaration:

interface Queue<E>

Here, E specifies the type of objects that the queue will hold. The methods defined by Queue are listed in the table below.

| Method | Description |
|---|---|
| E element( ) | Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty. |
| boolean offer(E *obj*) | Attempts to add *obj* to the queue. Returns **true** if *obj* was added and **false** otherwise. |
| E peek( ) | Returns the element at the head of the queue. It returns **null** if the queue is empty. The element is not removed. |
| E poll( ) | Returns the element at the head of the queue, removing the element in the process. It returns **null** if the queue is empty. |
| E remove( ) | Removes the element at the head of the queue, returning the element in the process. It throws **NoSuchElementException** if the queue is empty. |

**TABLE 17-5** The Methods Defined by **Queue**

**Collections - The Queue Interface**
Methods throw a **ClassCastException** when an object is incompatible with the elements in the queue.

A **NullPointerException** is thrown if an attempt is made to store a null object and null elements are not allowed in the queue.

An **IllegalArgumentException** is thrown if an invalid argument is used.

An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length queue that is full.

A **NoSuchElementException** is thrown if an attempt is made to remove an element from an empty queue.

Other facilities provided by the queue are as follows.

Elements can only be removed from the head of the queue.

There are two methods that obtain and remove elements: **poll( )** and **remove( )**.
The difference between them is that **poll( )** returns null if the queue is empty, but **remove( )** throws an exception.

**Collections - The Queue Interface**

Third, there are two methods, **element( )** and **peek( )**, that reads the element but will not remove the element at the head of the queue.

The difference is that element( ) throws an exception if the queue is empty, but peek( ) returns null.

**offer( )** only attempts to add an element to a queue. Because some queues have a fixed length and might be full, offer( ) can fail.

**Collections - The Deque Interface**

It extends Queue and declares the behavior of a double-ended queue.

Double-ended queues can function as standard, first-in, first-out (FIFO) queues or as last-in, first-out (LIFO) stacks. Deque is a generic interface that has this declaration:

interface Deque<E>

Here, E specifies the type of objects that the deque will hold.

In addition to the methods that it inherits from Queue, Deque adds those methods summarized in the table below.

# Collections - The Deque Interface

| Method | Description |
|---|---|
| void addFirst(E *obj*) | Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space. |
| void addLast(E *obj*) | Adds *obj* to the tail of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space. |
| Iterator<E> descendingIterator( ) | Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator. |
| E getFirst( ) | Returns the first element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty. |
| E getLast( ) | Returns the last element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty. |
| boolean offerFirst(E *obj*) | Attempts to add *obj* to the head of the deque. Returns **true** if *obj* was added and **false** otherwise. Therefore, this method returns **false** when an attempt is made to add *obj* to a full, capacity-restricted deque. |
| boolean offerLast(E *obj*) | Attempts to add *obj* to the tail of the deque. Returns **true** if *obj* was added and **false** otherwise. |
| E peekFirst( ) | Returns the element at the head of the deque. It returns **null** if the deque is empty. The object is not removed. |
| E peekLast( ) | Returns the element at the tail of the deque. It returns **null** if the deque is empty. The object is not removed. |

# Collections - The Deque Interface

| | |
|---|---|
| E pollFirst( ) | Returns the element at the head of the deque, removing the element in the process. It returns **null** if the deque is empty. |
| E pollLast( ) | Returns the element at the tail of the deque, removing the element in the process. It returns **null** if the deque is empty. |
| E pop( ) | Returns the element at the head of the deque, removing it in the process. It throws **NoSuchElementException** if the deque is empty. |
| void push(E *obj* ) | Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space. |
| E removeFirst( ) | Returns the element at the head of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty. |
| boolean removeFirstOccurrence(Object *obj*) | Removes the first occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*. |
| E removeLast( ) | Returns the element at the tail of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty. |
| boolean removeLastOccurrence(Object *obj*) | Removes the last occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*. |

**TABLE 17-6** The Methods Defined by **Deque**

**Collections - The DeQueue Interface**

Several methods throw a **ClassCastException** when an object is incompatible with the elements in the deque.

A **NullPointerException** is thrown if an attempt is made to store a null object and null elements are not allowed in the deque.

An **IllegalArgumentException** is thrown if an invalid argument is used.

An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length deque that is full.

A **NoSuchElementException** is thrown if an attempt is made to remove an element from an empty deque.

Notice that Deque includes the methods **push( )** and **pop( )**.

These methods enable a Deque to function as a stack.

The **descendingIterator( )** method, returns an iterator that returns elements in reverse order. In other words, it returns an iterator that moves from the end of the collection to the start.

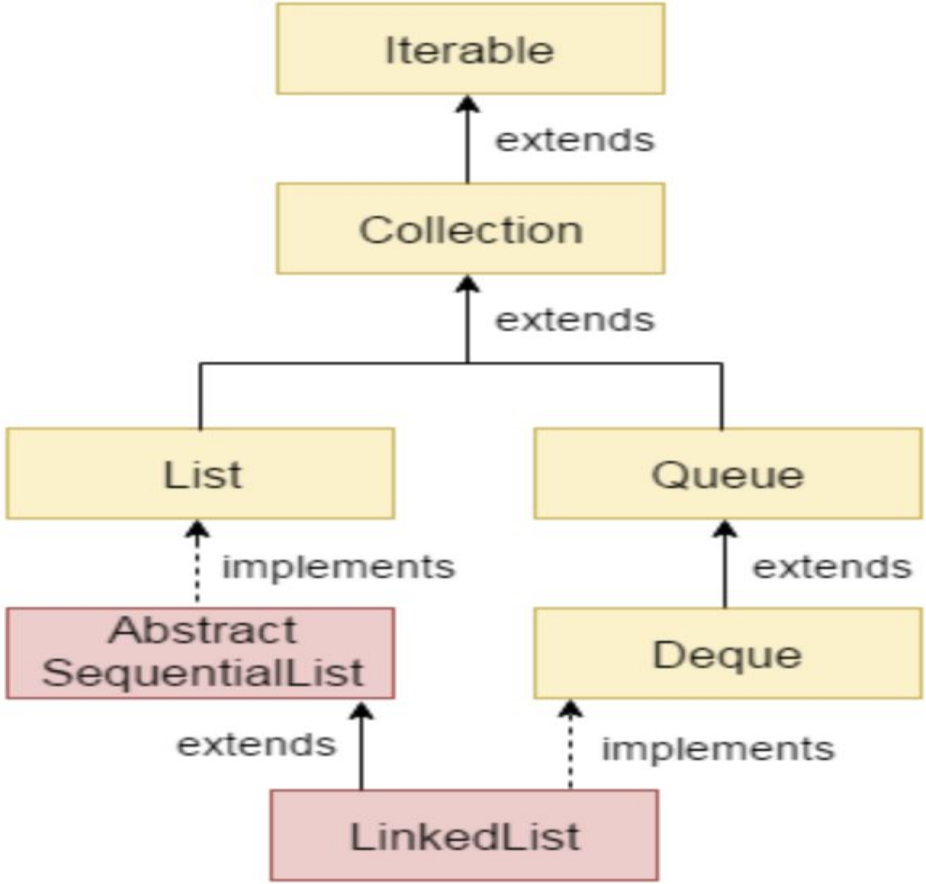
## Collections - The DeQueue Interface

A Deque implementation can be *capacity-restricted,* which means that only a limited number of elements can be added to the deque. When this is the case, an attempt to add an element to the deque can fail.

**Deque** allows programmers to handle such a failure in two ways.
First, methods such as **addFirst( )** and **addLast( )** throw an **IllegalStateException** if a capacity-restricted deque is full.

Second, methods such as **offerFirst( )** and **offerLast( )** return false if the element can not be added.

# Collections - The LinkedList



**Collections - The LinkedList Class**

The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interfaces.

It provides a linked-list data structure.

LinkedList is a generic class that has this declaration: **class LinkedList<E>**
Here, **E** specifies the type of objects that the list will hold.

LinkedList has the two constructors

**LinkedList( )**
**LinkedList(Collection<? extends E> c)**

The first constructor builds an empty linked list.

The second constructor builds a linked list that is initialized with the elements of the collection c.

Because **LinkedList** implements the **Deque** interface, you have access to the methods defined by **Deque**.

To add elements to the start of a list methods **addFirst( )** or **offerFirst( )**.

**Collections - The LinkedList Class**

To add elements to the end of the list methods **addLast( )** or **offerLast( )** can be used.

To obtain the first element, methods **getFirst( )** or **peekFirst( )** can be used.

To obtain the last element, use **getLast( )** or **peekLast( )**.

To remove the first element, use **removeFirst( )** or **pollFirst( )**.

To remove the last element, use **removeLast( )** or **pollLast( )**.

Ex:// Demonstrate LinkedList.

```
import java.util.*;
class Main {
public static void main(String args[]) {
        // Create a linked list.
        LinkedList<String> ll = new LinkedList<String>();

        // Add elements to the linked list.
        ll.add("F");  ll.add("B");   ll.add("D");   ll.add("E");  ll.add("C");   ll.addLast("Z");
        ll.addFirst("A");   ll.add(1, "A2");
        System.out.println("Original contents of ll: " + ll);
```

## Collections - The LinkedList Class

```
        // Remove elements from the linked list.
        ll.remove("F");    ll.remove(2);
        System.out.println("Contents of ll after deletion: "+ ll);

        // Remove first and last elements.
        ll.removeFirst();  ll.removeLast();
        System.out.println("ll after deleting first and last: " + ll);

        // Get and set a value.
        String val = ll.get(2);
        ll.set(2, val + " Changed");
        System.out.println("ll after change: " + ll);
    }
}
```

## Collections - The LinkedList Class

Because **LinkedList** implements the **List** interface, calls to **add(E)** append items to the end of the list, as do calls to **addLast( )**.

To insert items at a specific location, use the **add(int, E)** form of **add( )**, as illustrated by the call to **add(1, "A2")** in the example.

The third element in ll is changed by employing calls to **get( )** and **set( )**.

To obtain the current value of an element, pass **get( )** the index at which the element is stored. To assign a new value to that index, pass **set( )** the index and its new value.

## Collections - The Set Interface

The Set interface defines a set. It extends Collection and declares the behavior of a collection that **does not allow duplicate elements**.

Therefore, the **add( )** method returns false if an attempt is made to add duplicate elements to a set. It does not define any additional methods of its own.

Set is a generic interface that has this declaration:

interface Set<E>

Here, E specifies the type of objects that the set will hold.

## Collections - The HashSet Class

**HashSet** extends **AbstractSet** and implements the **Set** interface.

It creates a collection that uses a **hash table for storage**.

HashSet is a generic class that has this declaration:

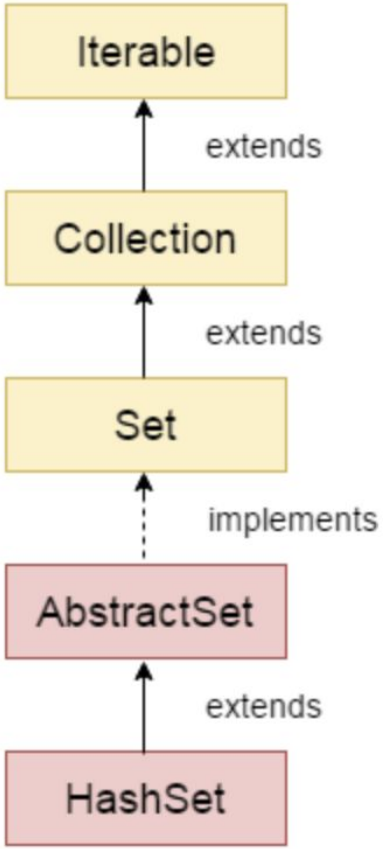
**class HashSet<E>**

Here, E specifies the type of objects that the set will hold.



A hash table stores information by using a mechanism called hashing.

# Collections - The HashSet Class



## Collections - The HashSet Class

In hashing, the informational content of a key is used to determine a unique value, called its hash code. The hash code is then used as the index at which the data associated with the key is stored.

The transformation of the key into its hashcode is performed automatically—hashcode is not known by the programmer, also the statements cannot index the hash table.

*The advantage of hashing is that it allows the execution time of* ***add( )****,* ***contains( )****,* ***remove( )****, and* ***size( )*** *to remain constant even for large sets.*

The following constructors are defined:

**HashSet( )**
**HashSet(Collection<? extends E> c)**
**HashSet(int capacity)**
**HashSet(int capacity, float fillRatio)**

The first form constructs a default hash set.

The second form initializes the hash set by using the elements of c.

The third form initializes the capacity of the hash set to capacity. (The default capacity is 16)

## Collections - The HashSet Class

The fourth form initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments.

The fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward.

Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded.

For constructors that do not take a fill ratio, 0.75 is used.

HashSet does not define any additional methods beyond those provided by its superclasses and interfaces.

It is important to note that **HashSet** does not guarantee the order of its elements, because the process of hashing doesn't usually lend itself to the creation of sorted sets.

## Collections - The HashSet Class

If sorted storage is required, then another collection, such as **TreeSet**, is a better choice.
Ex:

```
// Demonstrate HashSet.
import java.util.*;
class Main {
	public static void main(String args[]) {
		// Create a hash set.
		HashSet<String> hs = new HashSet<String>();

		// Add elements to the hash set.
		hs.add("B");    hs.add("A");  hs.add("D");  hs.add("E");  hs.add("C");  hs.add("F");

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

**Collections - The HashSet Class**

Ex:
```
import java.util.*;
class comp
{
    int i;
    public comp(int j)
    {  i=j;   }
    @Override
    public String toString()
    { return "obj "+i;  }

    @Override
    public boolean equals(Object obj)
    {
        System.out.println("In comp");
        comp t=(comp)obj;
```

## Collections - The LinkedHashSet Class

The **LinkedHashSet** class extends **HashSet** and adds no members of its own.

It is a generic class that has this declaration:

**class LinkedHashSet<E>**

Here, E specifies the type of objects that the set will hold.

Constructors are the same as in HashSet.

**LinkedHashSet** maintains a linked list of the entries in the set, in the order in which they were inserted. This allows insertion-order iteration over the set.

That is, when cycling through a **LinkedHashSet** using an iterator, the elements will be returned in the order in which they were inserted.

This is also the order in which they are contained in the string returned by **toString( )** when called on a **LinkedHashSet** object.

## Collections - The LinkedHashSet Class

Ex:

```
// Demonstrate HashSet.
import java.util.*;
class Main {
	public static void main(String args[]) {
		// Create a hash set.
		LinkedHashSet<String> hs = new LinkedHashSet<String>();

		// Add elements to the hash set.
		hs.add("B");
		hs.add("A");
		hs.add("D");
		hs.add("E");	hs.add("C");
		hs.add("F");
		System.out.println(hs);
	}
}
```

The output will be [B, A, D, E, C, F] which is the order in which the elements were inserted.

**Collections - The SortedSet Interface**

The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order.

SortedSet is a generic interface that has this declaration:

interface SortedSet<E>

Here, E specifies the type of objects that the set will hold.

The SortedSet interface present in java.util package extends the Set interface present in the collection framework.

It is an interface that implements the mathematical set.

In addition to those methods defined by Set, the SortedSet interface declares the methods summarized in the table below.

# Collections - The SortedSet Interface

| Method | Description |
|---|---|
| Comparator<? super E> comparator( ) | Returns the invoking sorted set's comparator. If the natural ordering is used for this set, **null** is returned. |
| E first( ) | Returns the first element in the invoking sorted set. |
| SortedSet<E> headSet(E *end*) | Returns a **SortedSet** containing those elements less than *end* that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set. |
| E last( ) | Returns the last element in the invoking sorted set. |
| SortedSet<E> subSet(E *start*, E *end*) | Returns a **SortedSet** that includes those elements between *start* and *end*–1. Elements in the returned collection are also referenced by the invoking object. |
| SortedSet<E> tailSet(E *start*) | Returns a **SortedSet** that contains those elements greater than or equal to *start* that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object. |

**TABLE 17-3** The Methods Defined by **SortedSet**

## Collections - The SortedSet Interface

Several methods throw a **NoSuchElementException** when no items are contained in the invoking set.

A **ClassCastException** is thrown when an object is incompatible with the elements in a set.

A **NullPointerException** is thrown if an attempt is made to use a null object and null is not allowed in the set.

An **IllegalArgumentException** is thrown if an invalid argument is used.

**SortedSet** defines several methods that make set processing more convenient.

To obtain the first object in the set, call **first( )**.

To get the last element, use **last( )**.

A subset of a sorted set can be obtained by calling **subSet( )**, specifying the first and last object in the set.

If a subset is needed that starts with the first element in the set, use **headSet( )**.

If a subset is needed that ends with the last element in the set then use **tailSet( )**.

## Collections - The NavigableSet Interface

It extends SortedSet and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values.

NavigableSet is a generic interface that has this declaration:

interface NavigableSet<E>

Here, E specifies the type of objects that the set will hold.

In addition to the methods that it inherits from SortedSet, NavigableSet adds those summarized in the table below.

**ClassCastException** is thrown when an object is incompatible with the elements in the set.

A **NullPointerException** is thrown if an attempt is made to use a null object and null is not allowed in the set.

An **IllegalArgumentException** is thrown if an invalid argument is used.

# Collections - The NavigableSet Interface

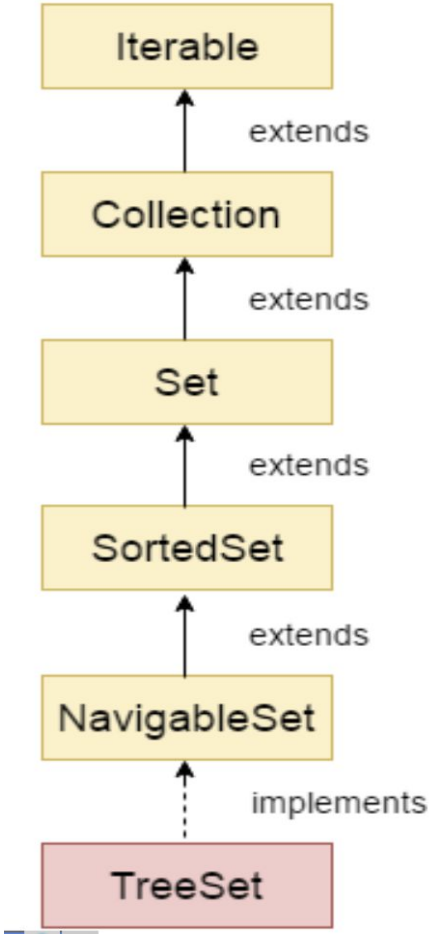
| Method | Description |
|---|---|
| E ceiling(E *obj*) | Searches the set for the smallest element *e* such that *e* >= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned. |
| Iterator<E> descendingIterator( ) | Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator. |
| NavigableSet<E> descendingSet( ) | Returns a **NavigableSet** that is the reverse of the invoking set. The resulting set is backed by the invoking set. |
| E floor(E *obj*) | Searches the set for the largest element *e* such that *e* <= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned. |
| NavigableSet<E> headSet(E *upperBound*, boolean *incl*) | Returns a **NavigableSet** that includes all elements from the invoking set that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set. |
| E higher(E *obj*) | Searches the set for the largest element *e* such that *e* > *obj*. If such an element is found, it is returned. Otherwise, **null** is returned. |
| E lower(E *obj*) | Searches the set for the largest element *e* such that *e* < *obj*. If such an element is found, it is returned. Otherwise, **null** is returned. |

# Collections - The NavigableSet Interface

| | |
|---|---|
| E pollFirst( ) | Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. **null** is returned if the set is empty. |
| E pollLast( ) | Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. **null** is returned if the set is empty. |
| NavigableSet<E><br>subSet(E *lowerBound*,<br>boolean *lowIncl*,<br>E *upperBound*,<br>boolean *highIncl*) | Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set. |
| NavigableSet<E><br>tailSet(E *lowerBound*, boolean *incl*) | Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound*. If *incl* is **true**, then an element equal to **lowerBound** is included. The resulting set is backed by the invoking set. |

**TABLE 17-4** The Methods Defined by **NavigableSet**

# Collections - The TreeSet Class



7. Explain with a suitable example how TreeSet class can be used to maintain instances of user defined data types in an order upon implementing Comparable interface.

## Collections - The TreeSet Class

**TreeSet** extends **AbstractSet** and implements the **NavigableSet** interface.

It creates a collection that uses a tree for storage. **Objects are stored in sorted, ascending order**.

Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

TreeSet is a generic class that has this declaration:

**class TreeSet<E>**

Here, E specifies the type of objects that the set will hold.

**TreeSet** has the following constructors:

**TreeSet( )**
**TreeSet(Collection<? extends E> c)**
**TreeSet(Comparator<? super E> comp)**
**TreeSet(SortedSet<E> ss)**

The first form constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements.

The second form builds a tree set that contains the elements of c.

The third form constructs an empty tree set that will be sorted according to the comparator specified by comp. (Comparators provide the sorting sequence for the information stored in TreeSet).

The fourth form builds a tree set that contains the elements of ss.

Ex:
```
class Main {
public static void main(String args[]) {
    // Create a tree set.
    TreeSet<String> ts = new TreeSet<String>();

    // Add elements to the tree set.
    ts.add("C");   ts.add("A");     ts.add("B");
    ts.add("E");   ts.add("F");     ts.add("D");
    System.out.println(ts);          } }
```

The output from this program is shown here: [A, B, C, D, E, F] Sorted order is maintained

## Collections - The TreeSet Class

While trying to store user defined types in TreeSet, the elements in TreeSet must be of a Comparable type.

Ex: If an attempt is made to store comp, user-defined type of information in TreeSet, comp class must implement **Comparable interface**.

**String** and **Wrapper** classes are Comparable by default.

Java Comparable interface is used to order the objects of the user-defined class.

Comparable interface is found in java.lang package and contains only one method named **compareTo(Object)**.

It provides a single sorting sequence only, i.e., the elements can be sorted on the basis of single data member only.

For example, it may be rollno, name, age or anything else.

## Collections - The TreeSet Class

**public int compareTo(Object obj):** It is used to compare the current object with the specified object. It returns

- positive integer, if the current object is greater than the specified object.
- negative integer, if the current object is less than the specified object.
- zero, if the current object is equal to the specified object.

## Collections - The TreeSet Class

```
import java.util.*;
class comp implements Comparable<comp>   {
        private int r, i;
        public comp(int a, int b)        {  r=a;  i=b;     }

        @Override
        public String toString()         { return r+" +i"+ i;  }

        @Override
        public int compareTo(comp b)  {
                if (r > b.r) return 1;
                if (r < b.r) return -1;
                return 0;
        }
}
```

## Collections - The TreeSet Class

```
public class First {
    public static void main(String[] args) {
        TreeSet<comp> a = new TreeSet<comp>();
        comp d = new comp(2,4);
        ts.add(new complex(8,9));        ts.add(new complex(3,4));
        ts.add(new complex(6,7));        ts.add(new complex(1,2));
        System.out.println(a.ceiling(d));  // prints 3+i4;
        comp f = new comp(5,6);
        System.out.println(a.floor(f));  // prints 3+i4;

    System.out.println(a.subSet(new complex(1,1), true, new complex(6,6), true));
        System.out.println(a);
    }
}
```

## Collections - The TreeSet Class

8. Compare and contrast Comparable and Comparator interfaces used in Java.

Comparable and Comparator both are interfaces and can be used to sort collection elements.

| **Comparable (java.lang)** | **Comparator (java.util)** |
|---|---|
| Comparable provides a **single sorting sequence**.<br>A collection can be sorted on the basis of a single element such as id, name, and price. | The Comparator provides **multiple sorting sequences**.<br>A collection can be sorted on the basis of multiple elements such as id, name, and price etc. |
| Comparable **affects the original class**, i.e., function is defined as a part of the class. | Comparator **doesn't affect the original class**, i.e., function is defined in a different class. |
| Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort elements. |
| We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)**method. |

## Collections - The TreeSet Class

```
import java.util.*;
class comp implements Comparable<comp>  {
        public int r, i, k;
        public comp(int a, int b, int c)    {  r=a; i=b; k=c;}
        public String toString()    {   return r+" "+i+" "+k;   }
        public int compareTo(comp b) //arranging objects on 'k' value, descending order
        {     int v;
              if (k > b.k)  v= -1;
// -1 returned, new object will be inserted to the front end of the list
              else
              if (k < b.k)  v=1;
// 1 returned, new object will be inserted to the rear end of the list
              else
              v=0;  // discarded because set does not maintain duplicate values.
              System.out.println(k + " " + b.k + " v " + v);
              return v;}      }
```

**Collections - The TreeSet Class**

```
class TheComparator implements Comparator<comp> {
    public int compare(comp p, comp q)    //Arranging objects based on 'r' value
    {
        if (p.r > q.r) return 1;
        if (p.r < q.r) return -1;
        return 0;
    }
}//end of class TheComparator
class TheComparator1 implements Comparator<comp>  {
    public int compare(comp p,comp q)  //Arranging objects based on 'i' value
    {
        if (p.i > q.i) return 1;
        if (p.i < q.i) return -1;
        return 0;
    }  }
```

## Collections - The TreeSet Class

```
public class First {
    public static void main(String[] args) {
        TreeSet<comp> a = new TreeSet<comp>(new TheComparator());
        a.add(new comp(1,2,3));
        a.add(new comp(0,4,4));
        a.add(new comp(-1,4,5));
        System.out.println("Sorting based on r value\n"+a);


        TreeSet<comp> b = new TreeSet<comp>(new TheComparator1());
        b.add(new comp(1,2,3));
        b.add(new comp(0,4,5));
        b.add(new comp(-1,-1,4));
        System.out.println("Sorting based on i value\n"+b);
```

## Collections - The TreeSet Class

```
        TreeSet<comp> c = new TreeSet<comp>();
        c.add(new comp(1,2,5));        c.add(new comp(0,4,4));
        c.add(new comp(-1,-1,3));
        System.out.println("Sorting based on k value\n"+c);

        LinkedList<comp> d = new LinkedList<comp>( );
        d.add(new comp(1,-4,3));       d.add(new comp(-1,-2,-3));
        d.add(new comp(0,0,0));

        Collections.sort(d);
        System.out.println(d);

        Collections.sort(d,new TheComparator1());
        System.out.println(d);
    }   }
```

## Collections - The PriorityQueue

9. Explain with suitable example when the priority is considered by the PriorityQueue class in Java

**PriorityQueue** extends **AbstractQueue** and implements the **Queue** interface.

It creates a queue that is prioritized based on the queue comparator.

**The priority of the elements in the PriorityQueue determine the order in which elements are removed from it.(***)**

**PriorityQueue** is a generic class that has this declaration:

**class PriorityQueue<E>** Here, E specifies the type of objects stored in the queue.

**PriorityQueue** are dynamic queues. PriorityQueue is based on priority heap.

**PriorityQueue** defines the six constructors shown here:

**PriorityQueue( )**
**PriorityQueue(int capacity)**
**PriorityQueue(int capacity, Comparator<? super E> comp)**
**PriorityQueue(Collection<? extends E> c)**
**PriorityQueue(PriorityQueue<? extends E> c)**
**PriorityQueue(SortedSet<? extends E> c)**

**Collections - The PriorityQueue**

The first constructor builds an empty queue. Its starting capacity is 11.

The second constructor builds a queue that has the specified initial capacity.

The third constructor builds a queue with the specified capacity and comparator.

The last three constructors create queues that are initialized with the elements of the collection passed in c.

In all cases, the capacity grows automatically as elements are added.

If no comparator is specified when a **PriorityQueue** is constructed, then the default comparator for the type of data stored in the queue is used.

The default comparator will order the queue in ascending order. Thus, the head of the queue might be the smallest value.

## Collections - The PriorityQueue

By providing a custom comparator, different ordering schemes can be specified.

A reference can be obtained to the comparator used by a PriorityQueue by calling its comparator( ) method:

**Comparator<? super E> comparator( )**

It returns the comparator.

If natural ordering is used for the invoking queue, null is returned.

Although a **PriorityQueue** can be iterated using an iterator, the order of that iteration is undefined.

To properly use a PriorityQueue, methods such as **offer( )** and **poll( )** must be called, which are defined by the **Queue** interface.

## Collections - The PriorityQueue

```
import java.util.*;
class comp implements Comparable<comp>  {
        public int r, i, k;
        public comp(int a, int b,int c)
        {  r=a;i=b;k=c;}

        @Override
        public String toString()  { return r+" "+i+" "+k; }

        @Override
        public int compareTo(comp b) //arranging objects on ‘k’ value
        {
              if (k > b.k) return 1;
              if (k < b.k) return -1;
              return 0; }      }
```

## Collections - The PriorityQueue

```
class TheComparable implements Comparator<comp>      {
        public int compare(comp a, comp b)    {
                if (a.r > b.r)return 1;
                if (a.r < b.r)return -1;
                return 0;                              }          }
public class First {
        public static void main(String[] args) {
                PriorityQueue<comp> c = new PriorityQueue();
                c.add(new comp(1,2,5));       c.add(new comp(0,4,4));
                c.add(new comp(-1,-1,3));     c.add(new comp(-1,-1,0));

                System.out.println("Sorting based on k value\n"+c);
                Iterator<comp> d = c.iterator();
                while(d.hasNext())
                        System.out.println(c.poll());    //displays contents in an order
```

## Collections - The PriorityQueue

```
        PriorityQueue<comp> b = new PriorityQueue(3, new TheComparable());
        b.add(new comp(1,2,5));        b.add(new comp(-9,4,4));
        b.add(new comp(100,-1,3));   b.add(new comp(-1,-1,0));

        System.out.println("Sorting based on r value\n"+b);  // no order while displaying

        Iterator<comp> d1 = b.iterator();
        while(d1.hasNext())
              System.out.println(b.poll()); //displays contents in an order

        Comparator<comp>z = (Comparator<comp>) b.comparator();
        System.out.println(z);
    }
}
```

## Collections - The ArrayDeque

**ArrayDeque** in Java provides a way to apply resizable-array in addition to the implementation of the Deque interface (slide 67)

**ArrayDeque** class, extends **AbstractCollection** and implements the **Deque** interface.

ArrayDeque is a generic class that has this declaration:

**class ArrayDeque<E>** Here, E specifies the type of objects stored in the collection.

ArrayDeque defines the following constructors:

**ArrayDeque( )** **ArrayDeque(int size)**

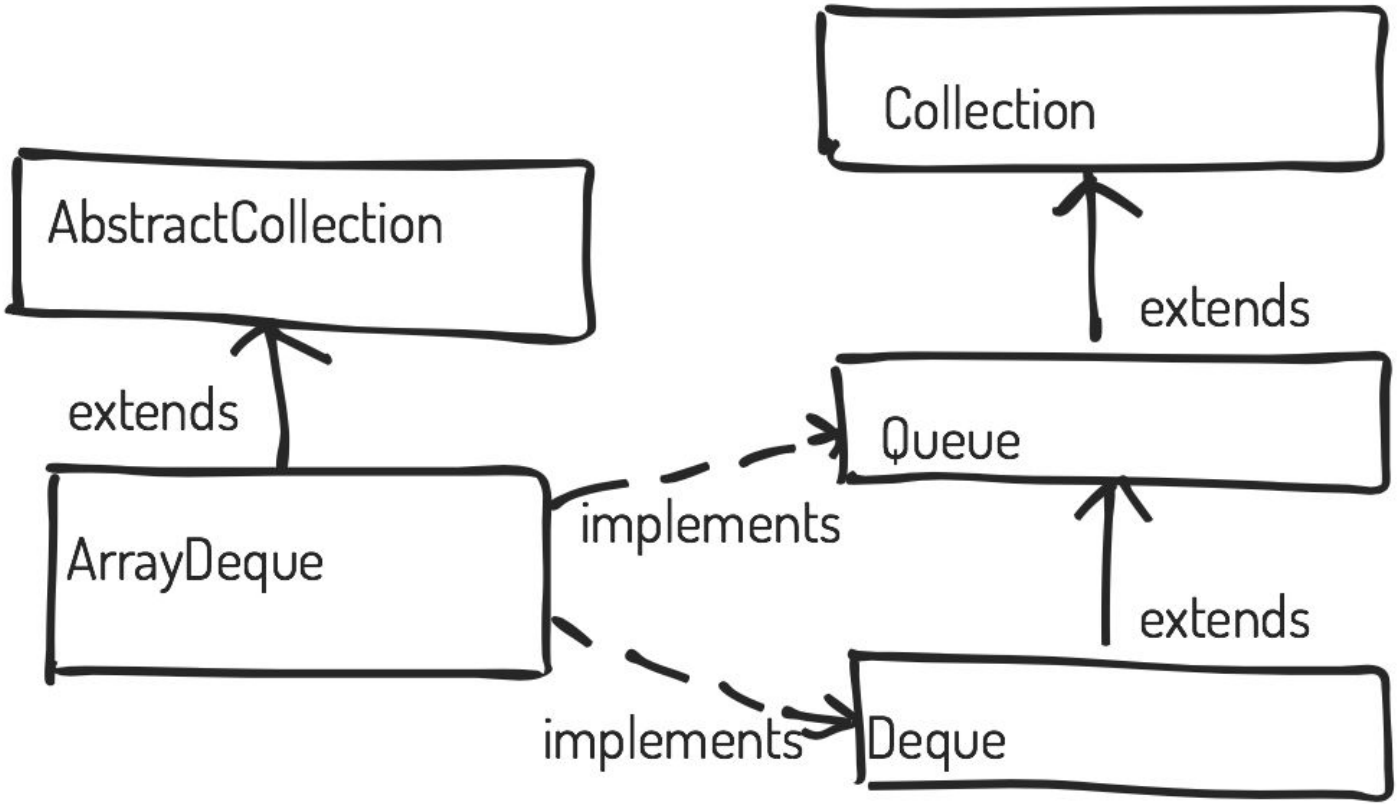
**ArrayDeque(Collection<? extends E> c)**

The first constructor builds an empty deque. Its starting capacity is 16.

The second constructor builds a deque that has the specified initial capacity.

The third constructor creates a deque that is initialized with the elements of the collection passed in c. In all cases, the capacity grows as needed to handle the elements added to the deque.

# Collections - The ArrayDeque



## Collections - The ArrayDeque

**ArrayDeque** class implements 2 interfaces

**Queue interface** : It is an Interface which confines to First-in First-out Data Structure where the elements are added from the rear end.

**Deque Interface:** It is a Doubly Ended Queue in which elements can be inserted from both the sides. It is an interface that implements the Queue.

ArrayDeque implements both Queue and Deque. It is dynamically resizable from front and rear sides of the queue.

## Collections - The ArrayDeque

### java.util.AbstractCollection

This class provides a skeletal implementation of the Collection interface, to minimize the effort required to implement this interface.

To implement an **unmodifiable collection**, the programmer needs only to extend this class and provide implementations for the iterator and size methods.

To implement a **modifiable collection**, the programmer must additionally override this class's add method (which otherwise throws an UnsupportedOperationException).

## Collections - The ArrayDeque

Elements can be **added** to ArrayDeque, by the following member functions

add( ), addFirst( ), addLast( ), offer( ), offerFirst( ) and offerLast( )

offer( ) and offerLast( ) adds element to the end of ArrayDeque

Elements can be **accessed**, by the following member functions

getFirst( ), getLast( ), peek( ), peekFirst( ), peekLast( )

peek( ) and peekLast( ) retrieves the last element

Elements can be **removed**, by the following member functions

remove( ), removeFirst( ), removeLast( ), poll( ), pollFirst( ), pollLast(), pop( );

**Collections - The ArrayDeque**

Elements can be **iterated**, by the following member functions
iterator( ), descendingIterator( )

**Collections - The EnumSet**

**EnumSet** extends **AbstractSet** and implements **Set**.

It is specifically used with keys/values of an enum type.

It is a generic class that has this declaration:

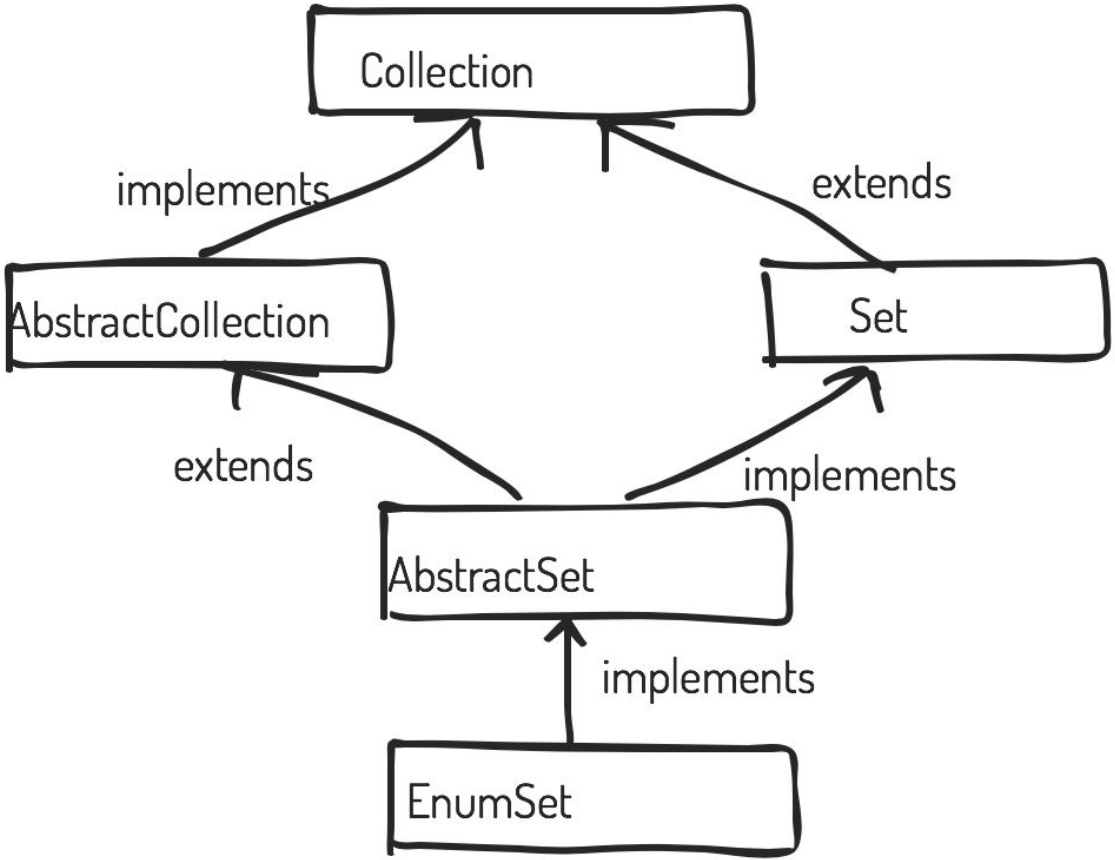
**class EnumSet<E extends Enum<E>>**

E specifies the elements, and E must extend Enum<E>,
which enforces the requirement that the elements must be of the specified enum type.

java.lang.Enum<E> is the common base class of all Java language enumeration types

EnumSet defines no constructors. Instead, it uses the static methods shown in the following table to create objects.

# Collections - The EnumSet



# Collections - The EnumSet

| Method | Description |
|---|---|
| static <E extends Enum<E>> EnumSet<E> allOf(Class<E> *t*) | Creates an **EnumSet** that contains the elements in the enumeration specified by *t*. |
| static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> *e*) | Creates an **EnumSet** that is comprised of those elements not stored in *e*. |
| static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> *c*) | Creates an **EnumSet** from the elements stored in *c*. |
| static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> *c*) | Creates an **EnumSet** from the elements stored in *c*. |
| static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> *t*) | Creates an **EnumSet** that contains the elements that are not in the enumeration specified by *t*, which is an empty set by definition. |
| static <E extends Enum<E>> EnumSet<E> of(E *v*, E ... *varargs*) | Creates an **EnumSet** that contains *v* and zero or more additional enumeration values. |
| static <E extends Enum<E>> EnumSet<E> of(E *v*) | Creates an **EnumSet** that contains *v*. |
| static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*) | Creates an **EnumSet** that contains *v1* and *v2*. |
| static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*) | Creates an **EnumSet** that contains *v1* through *v3*. |
| static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*) | Creates an **EnumSet** that contains *v1* through *v4*. |
| static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*, E *v5*) | Creates an **EnumSet** that contains *v1* through *v5*. |
| static <E extends Enum<E>> EnumSet<E> range(E *start*, E *end*) | Creates an **EnumSet** that contains the elements in the range specified by *start* and *end*. |

## Collections - The EnumSet

```
public class First {
    enum color {Red, Green, Blue, Violet, Pink, Black, White };

    public static void main(String[] args) {
        EnumSet<color> c = EnumSet.of (color.Red,  color.Green);
// creating an EnumSet collection with only two values from enum color

        System.out.println(c);

        EnumSet<color> d = EnumSet.noneOf(color.class);
// creating an EnumSet collection with noneOf the values in EnumSet c.

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

## Collections - The EnumSet

```
EnumSet<color> e = EnumSet.allOf(color.class);
//Adds all of the enum constants to the instance d of type EnumSet.

EnumSet<color> f = EnumSet.of(color.Red,color.Green);
EnumSet<color> g = EnumSet.complementOf(c);
//d EnumSet will be the complement of the values present in EnumSet c.

EnumSet<color> h = EnumSet.copyOf(c);
//d will be the exact copy of c

    EnumSet<color> i = EnumSet.of(color.Red,color.Green);
    color [] a = {color.Violet, color.Pink,color.Red };
    EnumSet<color> j = EnumSet.of(color.Green, a);   // using varargs 'a'

    System.out.println(j);   // d will contain G, V, P and R
```

## Collections - The RandomAccess marker interface

The **RandomAccess** interface contains no members.

However, by implementing this interface, a collection signals that it supports efficient random access to its elements.

By checking for the **RandomAccess** interface, it can be determined at run time whether a collection is suitable for certain types of random access operations—especially as they apply to large collections.

**RandomAccess** is implemented by ArrayList.

# Collections - The RandomAccess marker interface

```
ArrayList<Integer> a = new ArrayList<Integer>();
a.add(10);  a.add(12);
TreeSet<Integer> b = new TreeSet<Integer>();
b.add(10);  b.add(12);


if (a instanceof RandomAccess)
        System.out.println(" ArrayList supports RandomAccess ");
if (b instanceof RandomAccess)
        System.out.println(" TreeSet supports RandomAccess ");

System.out.println("ArrayList Random access "+a.get(1));
```

**Maps**

A map is an object that stores associations between keys and values, termed as key-value pairs.

Given a key, its value can be found.  Both keys and values are objects.

The keys must be unique, but the values may be duplicated.

Some maps can accept a null key and null values.

Maps does not implement the **Iterable** interface.

Maps cannot be circled through using for-each style for loop.

A collection-view can be obtained for a map, which allows the use of either the for loop or an iterator.

# Collections - Usage of Maps ???

10. Explain the usage of Map data structure in the collection framework.

Maps are perfect to use for key-value association mapping such as dictionaries.

The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys. Some examples are:

A map of error codes and their descriptions.

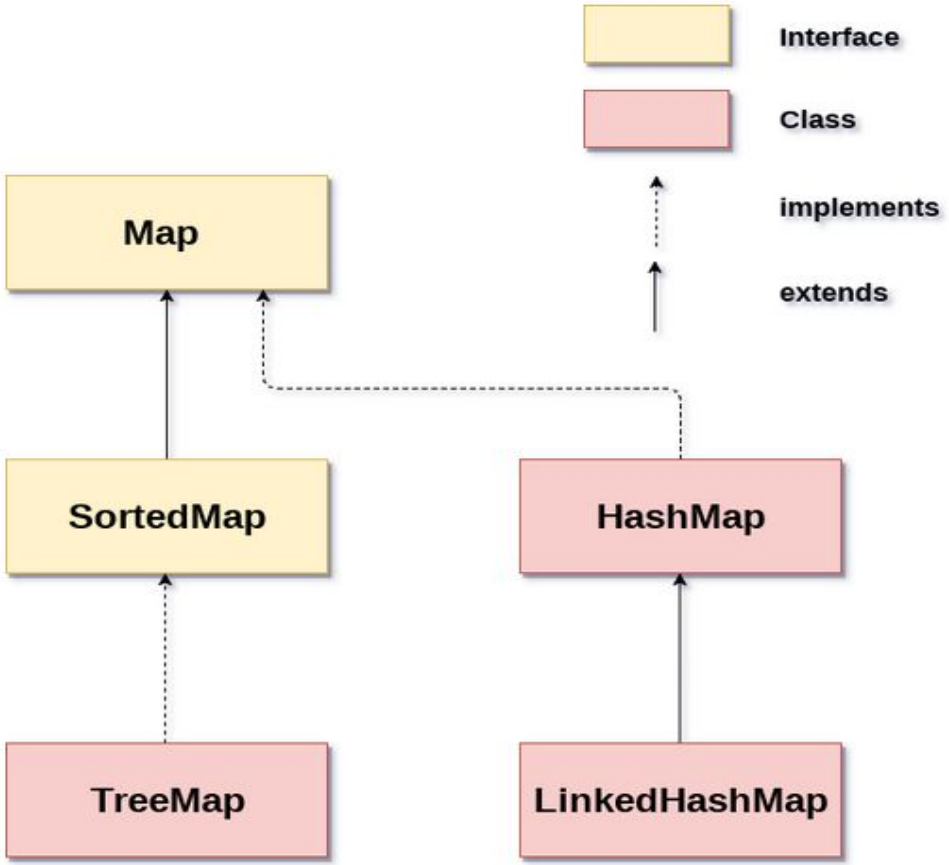
A map of zip codes and cities.

A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.

A map of classes and students. Each class (key) is associated with a list of students (value).

## Collections - The Map Interfaces

The map interfaces define the character and nature of maps.



**Collections - The Map Interfaces**

The following interfaces support maps:

| Interface | Description |
|---|---|
| Map | Maps unique keys to values. |
| Map.Entry | Describes an element (a key/value pair) in a map. This is an inner class of **Map**. |
| NavigableMap | Extends **SortedMap** to handle the retrieval of entries based on closest-match searches. (Added by Java SE 6.) |
| SortedMap | Extends **Map** so that the keys are maintained in ascending order. |

## Collections - The Map Interfaces

The **Map** interface maps unique keys to values.

Given a key and a value, it can be stored in a Map object.

Map is generic interface and is declared as:

**interface Map<K, V>**

Here, K specifies the type of keys, and V specifies the type of values.

The methods declared by Map are summarized in the table.

# Collections - The Map Interfaces

| Method | Description |
|---|---|
| void clear( ) | Removes all key/value pairs from the invoking map. |
| boolean containsKey(Object *k*) | Returns **true** if the invoking map contains *k* as a key. Otherwise, returns **false**. |
| boolean containsValue(Object *v*) | Returns **true** if the map contains *v* as a value. Otherwise, returns **false**. |
| Set<Map.Entry<K, V>> entrySet( ) | Returns a **Set** that contains the entries in the map. The set contains objects of type **Map.Entry**. Thus, this method provides a set-view of the invoking map. |
| boolean equals(Object *obj*) | Returns **true** if *obj* is a **Map** and contains the same entries. Otherwise, returns **false**. |
| V get(Object *k*) | Returns the value associated with the key *k*. Returns **null** if the key is not found. |
| int hashCode( ) | Returns the hash code for the invoking map. |
| boolean isEmpty( ) | Returns **true** if the invoking map is empty. Otherwise, returns **false**. |

# Collections - The Map Interfaces

| | |
|---|---|
| Set<K> keySet( ) | Returns a **Set** that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map. |
| V put(K *k*, V *v*) | Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are *k* and *v*, respectively. Returns **null** if the key did not already exist. Otherwise, the previous value linked to the key is returned. |
| void putAll(Map<? extends K, ? extends V> *m*) | Puts all the entries from *m* into this map. |
| V remove(Object *k*) | Removes the entry whose key equals *k*. |
| int size( ) | Returns the number of key/value pairs in the map. |
| Collection<V> values( ) | Returns a collection containing the values in the map. This method provides a collection-view of the values in the map. |

**TABLE 17-10** The Methods Defined by **Map**

## Collections - The Map Interfaces

Several methods throw a **ClassCastException** when an object is incompatible with the elements in a map.

A **NullPointerException** is thrown if an attempt is made to use a null object and null is not allowed in the map.

An **UnsupportedOperationException** is thrown when an attempt is made to change an unmodifiable map.

An **IllegalArgumentException** is thrown if an invalid argument is used.

Maps support two basic operations: **get( )** and **put( )**.

To put a value into a map, use put( ), specifying the key and the value.

To obtain a value, call get( ), passing the key as an argument. The value is returned.

## Collections - The Map Interfaces

Although Maps are part of the Collections Framework, maps are not, themselves, collections because they **do not implement the Collection interface.**

However, a collection-view of a map can be obtained, by calling **entrySet( )** method.

It returns a **Set** that contains the elements in the map.

To obtain a collection-view of the keys, use **keySet( )**.

To get a collection-view of the values, use **values( )**.

**Collection-views are the means by which maps are integrated into the larger Collections Framework.**

## Collections - The SortedMap Interfaces

The **SortedMap** interface extends Map.

It ensures that the **entries are maintained in ascending order based on the keys**.

SortedMap is generic and is declared as shown here:

**interface SortedMap<K, V>**

Here, K specifies the type of keys, and V specifies the type of values.

The methods declared by **SortedMap** are summarized in the table below.

# Collections - The SortedMap Interfaces

| Method | Description |
|---|---|
| Comparator<? super K> comparator( ) | Returns the invoking sorted map's comparator. If natural ordering is used for the invoking map, **null** is returned. |
| K firstKey( ) | Returns the first key in the invoking map. |
| SortedMap<K, V> headMap(K *end*) | Returns a sorted map for those map entries with keys that are less than *end*. |
| K lastKey( ) | Returns the last key in the invoking map. |
| SortedMap<K, V> subMap(K *start*, K *end*) | Returns a map containing those entries with keys that are greater than or equal to *start* and less than *end*. |
| SortedMap<K, V> tailMap(K *start*) | Returns a map containing those entries with keys that are greater than or equal to *start*. |

**TABLE 17-11** The Methods Defined by **SortedMap**

## Collections - The SortedMap Interfaces

Several methods throw a **NoSuchElementException** when no items are in the invoking map.

A **ClassCastException** is thrown when an object is incompatible with the elements in a map.

A **NullPointerException** is thrown if an attempt is made to use a null object when null is not allowed in the map.

An **IllegalArgumentException** is thrown if an invalid argument is used.

Sorted maps allow very efficient manipulations of submaps (subsets of a map).

To obtain a submap, use **headMap( )**, **tailMap( )**, or **subMap( )**.

To get the first key in the set, **firstKey( )**.

To get the last key, **lastKey( )**.

## Collections - The NavigableMap Interfaces

**NavigableMap** extends **SortedMap** and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys.

NavigableMap is a generic interface that has this declaration:

**interface NavigableMap<K, V>**

Here, K specifies the type of the keys, and V specifies the type of the values associated with the keys.

In addition to the methods that it inherits from SortedMap, NavigableMap adds those summarized in the table below.

# Collections - The NavigableMap Interfaces

| Method | Description |
|---|---|
| Map.Entry<K,V> ceilingEntry(K *obj*) | Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned. |
| K ceilingKey(K *obj*) | Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned. |
| NavigableSet<K> descendingKeySet( ) | Returns a **NavigableSet** that contains the keys in the invoking map in reverse order. Thus, it returns a reverse set-view of the keys. The resulting set is backed by the map. |
| NavigableMap<K,V> descendingMap( ) | Returns a **NavigableMap** that is the reverse of the invoking map. The resulting map is backed by the invoking map. |
| Map.Entry<K,V> firstEntry( ) | Returns the first entry in the map. This is the entry with the least key. |
| Map.Entry<K,V> floorEntry(K *obj*) | Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned. |
| K floorKey(K *obj*) | Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned. |
| NavigableMap<K,V> headMap(K *upperBound*, boolean *incl*) | Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting map is backed by the invoking map. |
| Map.Entry<K,V> higherEntry(K *obj*) | Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned. |

**TABLE 17-12** The Methods defined by **NavigableMap**

# Collections - The NavigableMap Interfaces

| Method | Description |
|---|---|
| K higherKey(K *obj*) | Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, it is returned. Otherwise, **null** is returned. |
| Map.Entry<K,V> lastEntry( ) | Returns the last entry in the map. This is the entry with the largest key. |
| Map.Entry<K,V> lowerEntry(K *obj*) | Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned. |
| K lowerKey(K *obj*) | Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, it is returned. Otherwise, **null** is returned. |
| NavigableSet<K> navigableKeySet( ) | Returns a **NavigableSet** that contains the keys in the invoking map. The resulting set is backed by the invoking map. |
| Map.Entry<K,V> pollFirstEntry( ) | Returns the first entry, removing the entry in the process. Because the map is sorted, this is the entry with the least key value. **null** is returned if the map is empty. |
| Map.Entry<K,V> pollLastEntry( ) | Returns the last entry, removing the entry in the process. Because the map is sorted, this is the entry with the greatest key value. **null** is returned if the map is empty. |
| NavigableMap<K,V><br>subMap(K *lowerBound*,<br>boolean *lowIncl*,<br>K *upperBound*<br>boolean *highIncl*) | Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *highIncl* is included. The resulting map is backed by the invoking map. |
| NavigableMap<K,V><br>tailMap(K *lowerBound*, boolean *incl*) | Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting map is backed by the invoking map. |

**TABLE 17-12** The Methods defined by **NavigableMap** *(continued)*

**Collections - The NavigableMap Interfaces**

Several methods throw a **ClassCastException** when an object is incompatible with the keys in the map.

A **NullPointerException** is thrown if an attempt is made to use a null object and null keys are not allowed in the set.

An **IllegalArgumentException** is thrown if an invalid argument is used.

**Collections - The Map.Entry Interface**

The **entrySet( )** method declared by the **Map** interface returns a **Set** containing the map entries. Each of these set elements is a **Map.Entry** object.

Map.Entry is generic:

**interface Map.Entry<K, V>**

Here, K specifies the type of keys, and V specifies the type of values.

Table below summarizes the methods declared by **Map.Entry**.

# Collections - The Map.Entry Interface

| Method | Description |
|---|---|
| boolean equals(Object *obj*) | Returns **true** if *obj* is a **Map.Entry** whose key and value are equal to that of the invoking object. |
| K getKey( ) | Returns the key for this map entry. |
| V getValue( ) | Returns the value for this map entry. |
| int hashCode( ) | Returns the hash code for this map entry. |
| V setValue(V *v*) | Sets the value for this map entry to *v*. A **ClassCastException** is thrown if *v* is not the correct type for the map. An **IllegalArgumentException** is thrown if there is a problem with *v*. A **NullPointerException** is thrown if *v* is **null** and the map does not permit **null** keys. An **UnsupportedOperationException** is thrown if the map cannot be changed. |

**TABLE 17-13** The Methods Defined by **Map.Entry**

## Collections - The Map Class

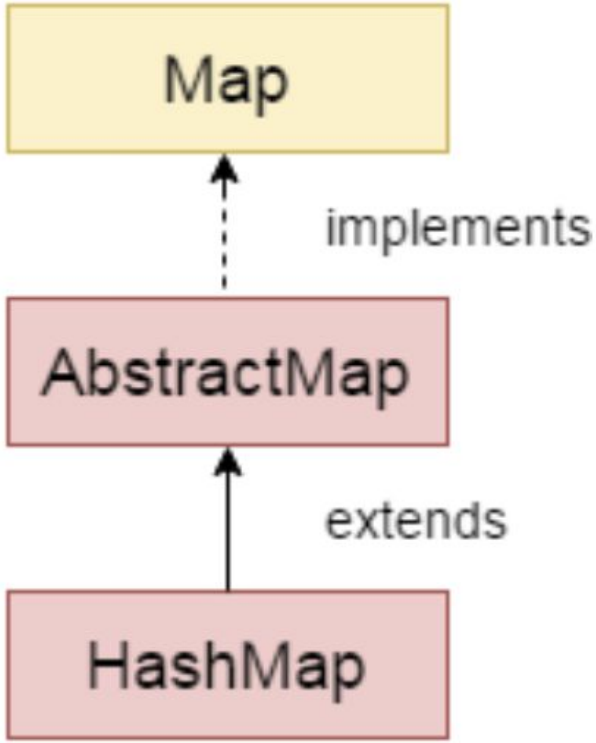
Several concrete classes provide implementations of the map interfaces.

The classes that can be used for maps are summarized here:

| Class | Description |
|---|---|
| AbstractMap | Implements most of the **Map** interface. |
| EnumMap | Extends **AbstractMap** for use with **enum** keys. |
| HashMap | Extends **AbstractMap** to use a hash table. |
| TreeMap | Extends **AbstractMap** to use a tree. |
| WeakHashMap | Extends **AbstractMap** to use a hash table with weak keys. |
| LinkedHashMap | Extends **HashMap** to allow insertion-order iterations. |
| IdentityHashMap | Extends **AbstractMap** and uses reference equality when comparing documents. |

**Collections - The HashMap Class**

HashMap has the implemented member functions of Map interface only.



## Collections - The HashMap Class

The **HashMap** class extends **AbstractMap** and implements the **Map** interface.

It uses a hash table to store the map.

This allows the execution time of get( ) and put( ) to remain constant even for large sets.

HashMap is a generic class that has this declaration:

**class HashMap<K, V>**

Here, K specifies the type of keys, and V specifies the type of values.

The following constructors are defined:

**HashMap( )**
**HashMap(Map<? extends K, ? extends V> m)**
**HashMap(int capacity)**
**HashMap(int capacity, float fillRatio)**

## Collections - The HashMap Class

The first form constructs a default hash map.

The second form initializes the hash map by using the elements of m.

The third form initializes the capacity of the hash map to capacity.

The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments.(meaning of capacity and fill ratio is the same as for **HashSet**)
The default capacity is 16. The default fill ratio is 0.75.

**HashMap** implements **Map** and extends **AbstractMap**.

It does not add any methods of its own. **Hash map will not guarantee the order of its elements.** Therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator.

## Collections - The HashMap Class

```
public class First {
    public static void main(String[] args) {
        // Create a hash map.
    HashMap<String, Double> hm = new HashMap<String, Double>();
    // Put elements to the map
    hm.put("JD", 3434.34);
    hm.put("TS", 123.22);
    hm.put("JB", 1378.00);
    hm.put("TH", 99.22);
    hm.put("RS", -19.08);

    System.out.println(hm.containsKey("TH"));
    System.out.println(hm.containsValue(-19.08));
```

## Collections - The HashMap Class

```
// Get a set of the entries.
Set<Map.Entry<String, Double>> set = hm.entrySet();

// Display the set.
for(Map.Entry<String, Double> me : set) {
        System.out.print(me.getKey() + ": ");
                System.out.println(me.getValue());
}

HashMap<String,Integer> r = new HashMap<String,Integer>();
r.put("A",1); r.put("B", 2);

HashMap<String,Integer> s = new HashMap<String,Integer>();
s.put("B", 2);  s.put("A",1);
System.out.println("equals "+r.equals(s));
```

## Collections - The HashMap Class

```
        System.out.println();
        // Deposit 1000 into John Doe's account.
        double balance = hm.get("JD");
        hm.put("JD", balance + 1000);

 System.out.println("JD new balance: " + hm.get("JD"));
                Map<String,Double> p = new HashMap<String,Double>();
                p.put("A", 100.09);
                p.put("B", 101.09);
                hm.putAll(p);
        }
}
```

## Collections - The TreeMap class

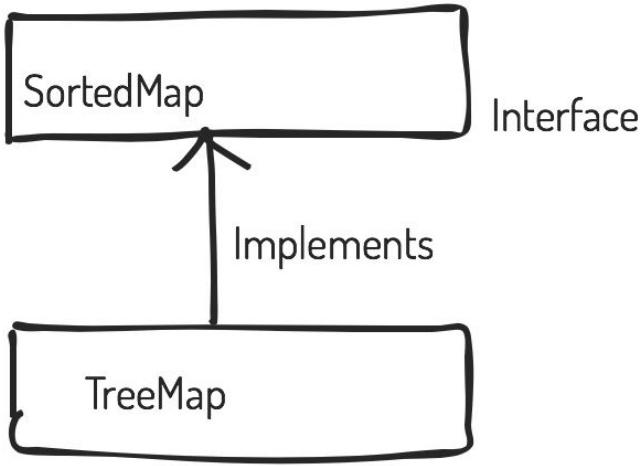
11. Explain the different types of constructors used in the TreeMap class.

The **TreeMap** class extends **AbstractMap** and implements the **NavigableMap** interface.

**It stores map informations in a tree structure**.

A TreeMap provides an efficient means of storing key/value pairs in **sorted order** and allows rapid retrieval.

A tree map guarantees that its elements will be sorted in ascending key order.



## Collections - The TreeMap class

TreeMap is a generic class that has this declaration:
**class TreeMap<K, V>** Here, K specifies the type of keys, and V specifies the type of values.

The following TreeMap constructors are defined:

**TreeMap( )** **TreeMap(Comparator<? super K> comp)**
**TreeMap(Map<? extends K, ? extends V> m)**
**TreeMap(SortedMap<K, ? extends V> sm)**

The first form constructs an empty tree map that will be sorted by using the natural order of its keys.

The second form constructs an empty tree-based map that will be sorted by using the **Comparator comp**.

The third form initializes a tree map with the entries from m, which will be sorted by using the natural order of the keys.

The fourth form initializes a tree map with the entries from sm, which will be sorted in the same order as sm.

## Collections - The TreeMap class

**TreeMap** has no methods beyond those specified by the **NavigableMap** interface and the **AbstractMap** class.

```
import java.util.Comparator;
import java.util.TreeMap;

class student implements Comparable<student>
{
      private String usn,name;
      public student(String u, String n)
      { usn = u; name = n;  }
      public String getusn() { return usn; }
      public String getname( ) { return name; }
      public String toString()   {      return usn+" "+name;      }
```

## Collections - The TreeMap class

```
        @Override
        public int compareTo(student o) {
                int v= usn.compareTo(o.usn);
                if (v>0) return 1;
                if (v==0) return 0;

                return -1;
        }     }
    class theComparator implements Comparator<student>       {
        @Override
        public int compare(student p, student q) {
                int v=(p.getname()).compareTo(q.getname());
                if (v>0) return 1;
                if (v==0) return 0;
                return -1;
        }
    }
```

## Collections - The TreeMap class

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

          TreeMap<student,Integer> t = new TreeMap<student,Integer>();
          t.put(new student("1RN19CS010","A"),1);
          t.put(new student("1RN19CS001","B"),2);
          t.put(new student("1RN19CS000","C"),3);
          System.out.println(t);

TreeMap<student,Integer> p = new TreeMap<student,Integer>(new theComparator());
          p.put(new student("1RN19CS010","C"),1);
          p.put(new student("1RN19CS001","B"),2);
          p.put(new student("1RN19CS000","A"),3);
          System.out.println(p);
    }
  }
```

**Collections - The LinkedHashMap class**

LinkedHashMap extends HashMap. It maintains a linked list of the entries in the map, **in the order in which they were inserted**.

This allows insertion-order iteration over the map.
That is, when iterating through a collection-view of a **LinkedHashMap**, the elements will be returned in the order in which they were inserted.

**LinkedHashMap** is a generic class that has this declaration:
**class LinkedHashMap<K, V>** Here, K specifies the type of keys, and V specifies the type of values.

LinkedHashMap defines the following constructors:

**LinkedHashMap( )**
**LinkedHashMap(Map<? extends K, ? extends V> m)**
**LinkedHashMap(int capacity)**
**LinkedHashMap(int capacity, float fillRatio)**
**LinkedHashMap(int capacity, float fillRatio, boolean Order)**

## Collections - The LinkedHashMap class

The first form constructs a default **LinkedHashMap**.

The second form initializes the **LinkedHashMap** with the elements from m.

The third form initializes the capacity.

The fourth form initializes both capacity and fill ratio. The meaning of capacity and fill ratio are the same as for HashMap. The default capacity is 16. The default ratio is 0.75.

The last form of constructor allows programmers to specify whether the elements will be stored in the linked list by insertion order, or by order of last access.

If Order is true, then access order is used. If Order is false, then insertion order is used.

**Collections - The LinkedHashMap class**

LinkedHashMap adds only one method to those defined by HashMap. This method is **removeEldestEntry( )** and it is shown here:

**protected boolean removeEldestEntry(Map.Entry<K, V> e)**

This method is called by put( ) and putAll( ). The oldest entry is passed in e.

**By default (generic)**, this method returns false and does nothing.

However, if this method is overridden then **LinkedHashMap** removes the oldest entry in the map.

**Access order & Insertion order**

Considering, the 4th form of constructor associated with LinkedHashMap:

The last parameter can be set to **true** or **false**.

If set to true, access order will be used, default is insertion order.

12. Explain how Access order and Insertion order are maintained by the LinkedHashMap class in Java with suitable code snippets

**Collections - Access order & Insertion order**

Access order ensures that the order-of-iteration of elements is the order in which the elements were last accessed, from Least-Recently accessed to Most-Recently accessed.

"Order of iteration" - explicit access of the map contents will affect the access order.

**Insertion order**

refers to the order in which elements are added to the collection framework.

Ex: a List object maintains the order in which elements are added, whereas a Set object doesn't maintain the order of the elements in which they are added.

Ex:
```
LinkedHashMap<Integer, String> u=null;
u = new LinkedHashMap<Integer, String>(4, 0.4f, true);  // Access order
u.put(3,"C"); u.put(1,"A"); u.put(2,"B");  u.put(4,"D");
System.out.println(u);      u.put(1,"F");
System.out.println(u);      u.put(3, "CC");
System.out.println(u);
```

**Collections - Access order & Insertion order**

Output will be

{3=C, 1=A, 2=B, 4=D}

{3=C, 2=B, 4=D, 1=F}

{2=B, 4=D, 1=F, 3=CC}

Access order has been considered by passing “true” as the last parameter, and insertion order is also maintained as can be seen in the 1st line of the output.

Redundant key value (1,F) is added which forces old entry (1,A) to be removed and (1, F) is added as the last entry. (2nd line output)

Existing key,value pair (3,C) is modified as (3,CC), hence it is displayed as the last value. (Access order - Least Recently used ---> Most Recently used.

**Collections - Access order & Insertion order**

```
LinkedHashMap<Integer, String> u=null;
u = new LinkedHashMap<Integer, String>(4,  0.4f,  false);
```

The last parameter is false, enforcing not to use access order, and insertion order is maintained.

```
u.put(3,"C"); u.put(1,"A"); u.put(2,"B");  u.put(4,"D");
System.out.println(u);    u.put(1,"F");
System.out.println(u);    u.put(3, "CC");
System.out.println(u);
```

Output: {3=C, 1=A, 2=B, 4=D}
{3=C, 1=F, 2=B, 4=D}
{3=CC, 1=F, 2=B, 4=D}

(1,A) value is modified by (1,F), 2nd line output

(3, C) value is modified by (3, CC), 3rd line output

Maintaining the insertion order.

**Collections - Access order & Insertion order on User defined types**

Ex:

```
class student {
        private String usn,name;
        public student(String u, String n)
        { usn = u; name = n;  }

        public String getusn() { return usn; }
        public String getname( ) { return name; }

        public String toString()
        {
              return usn+" "+name;
        }
}
```

**Collections - Access order & Insertion order on User defined types**

```
class First  {
  public static void main(String[] args) {
        LinkedHashMap<Integer,student> m = null;
        m = new LinkedHashMap<Integer,student>(4,  0.4f,  true);
        m.put(1, null);          m.put(2, null);          m.put(3, null);          m.put(4, null);
        System.out.println(m);          m.put(1,new student("A","1"));
        System.out.println(m);          m.put(4, new student("D","4"));
        System.out.println(m);
  }  }
```

Output:  {1=null, 2=null, 3=null, 4=null}
{2=null, 3=null, 4=null, 1=A 1}
{2=null, 3=null, 1=A 1, 4=D 4}

Access order maintained Least-to-Most recently used

## Collections - Access order & Insertion order on User defined types

```
class First  {
  public static void main(String[] args) {
        LinkedHashMap<Integer,student> m = null;
        m = new LinkedHashMap<Integer,student>(4,0.4f,false);
        m.put(1, null);          m.put(2, null);          m.put(3, null);          m.put(4, null);
        System.out.println(m);          m.put(1,new student("A","1"));
        System.out.println(m);          m.put(4, new student("D","4"));
        System.out.println(m);
  } }
```

Output:

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

{1=A 1, 2=null, 3=null, 4=null}

{1=A 1, 2=null, 3=null, 4=D 4}

Insertion order maintained

## Collections - Using 2nd form of LinkedHashMap constructor
## LinkedHashMap(Map<? extends K, ? extends V> m)

```
Map<Integer,student> q = new HashMap<Integer,student>();
q.put(1,new student("1RN18CS010","A"));
q.put(2,new student("1RN18CS001","B"));
q.put(3,new student("1RN18CS000","C"));

System.out.println(q);
LinkedHashMap<Integer,student> t = new LinkedHashMap<Integer,student>(q);
t.put(1,new student("1RN19CS010","A"));
t.put(2,new student("1RN19CS001","B"));
t.put(3,new student("1RN19CS000","C"));
System.out.println(t);
```

Output: {1=1RN18CS010 A, 2=1RN18CS001 B, 3=1RN18CS000 C}
{1=1RN19CS010 A, 2=1RN19CS001 B, 3=1RN19CS000 C}

**Collections - EnumMap class**

**EnumMap** extends **AbstractMap** and implements **Map**.

It is specifically, used with keys of an enum type.

It is a generic class that has this declaration:

**class EnumMap<K extends Enum<K>, V>**

Here, K specifies the type of key, and V specifies the type of value.

K must extend Enum<K>, which enforces the requirement that the keys must be of an enum type.

EnumMap defines the following constructors:

**EnumMap(Class<K> kType)**

**EnumMap(Map<K, ? extends V> m)**

**EnumMap(EnumMap<K, ? extends V> em)**

**Collections - EnumMap class**

The first constructor creates an empty EnumMap of type k-Type.

The second creates an EnumMap map that contains the same entries as m.

The third creates an EnumMap initialized with the values in em.

EnumMap defines no methods of its own.

**EnumMap(Class<K> kType)** first constructor

```
Ex: enum days { Sun, Mon, Tue, Wed, Thur };
    EnumMap<days,Integer> p = new EnumMap<days,Integer>(days.class);
    p.put(days.Sun, 1);  p.put(days.Mon, 2);  p.put(days.Tue, 3);
    System.out.println(p);  p.put(days.Sun, 0);
     System.out.println(p);
```

```
Output: {Sun=1, Mon=2, Tue=3}
        {Sun=0, Mon=2, Tue=3}
```

## Collections - EnumMap class

**EnumMap(Map<K, ? extends V> m)**

Ex:

```
enum days  { Sun, Mon, Tue, Wed, Thur };
Map<days,Integer> m = new LinkedHashMap<days,Integer>();
m.put(days.Sun,1);    m.put(days.Mon,2);
m.put(days.Tue,3);
EnumMap<days,Integer> p = new EnumMap(m);
System.out.println(p);
```

**EnumMap(EnumMap<K, ? extends V> em)**

Ex:

```
EnumMap<days,Integer> q = new EnumMap(p);
System.out.println(q);
```

## Collections - The Collection Algorithm

The Collections Framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the **Collections** class.

| Method | Description |
|---|---|
| static <T> boolean<br>addAll(Collection <? super T> *c*,<br>T ... *elements*) | Inserts the elements specified by *elements* into the collection specified by *c*. Returns **true** if the elements were added and **false** otherwise. |
| static <T> Queue<T> asLifoQueue(Deque<T> *c*) | Returns a last-in, first-out view of *c*. (Added by Java SE 6.) |
| static <T><br>int binarySearch(List<? extends T> *list*,<br>T *value*,<br>Comparator<? super T> *c*) | Searches for *value* in *list* ordered according to *c*. Returns the position of *value* in *list*, or a negative value if *value* is not found. |
| static <T><br>int binarySearch(List<? extends<br>Comparable<? super T>> *list*,<br>T *value*) | Searches for *value* in *list*. The list must be sorted. Returns the position of *value* in *list*, or a negative value if *value* is not found. |
| static <E> Collection<E><br>checkedCollection(Collection<E> *c*,<br>Class<E> *t*) | Returns a run-time type-safe view of a collection. An attempt to insert an incompatible element will cause a **ClassCastException**. |
| static <E> List<E><br>checkedList(List<E> *c*, Class<E> *t*) | Returns a run-time type-safe view of a **List**. An attempt to insert an incompatible element will cause a **ClassCastException**. |
| static <K, V> Map<K, V><br>checkedMap(Map<K, V> *c*,<br>Class<K> *keyT*,<br>Class<V> *valueT*) | Returns a run-time type-safe view of a **Map**. An attempt to insert an incompatible element will cause a **ClassCastException**. |
| static <E> List<E><br>checkedSet(Set<E> *c*, Class<E> *t*) | Returns a run-time type-safe view of a **Set**. An attempt to insert an incompatible element will cause a **ClassCastException**. |

# Collections - The Collection Algorithm

| | |
|---|---|
| static <K, V> SortedMap<K, V><br>checkedSortedMap(SortedMap<K, V> *c*,<br>Class<K> *keyT*,<br>Class<V> *valueT*) | Returns a run-time type-safe view of a **SortedMap**. An attempt to insert an incompatible element will cause a **ClassCastException**. |
| static <E> SortedSet<E><br>checkedSortedSet(SortedSet<E> *c*, Class<E> *t*) | Returns a run-time type-safe view of a **SortedSet**. An attempt to insert an incompatible element will cause a **ClassCastException**. |
| static <T> void copy(List<? super T> *list1*,<br>List<? extends T> *list2*) | Copies the elements of *list2* to *list1*. |
| static boolean disjoint(Collection<?> *a*,<br>Collection<?> *b*) | Compares the elements in *a* to elements in *b*. Returns **true** if the two collections contain no common elements (i.e., the collections contain disjoint sets of elements). Otherwise, returns **true**. |
| static <T> List<T> emptyList( ) | Returns an immutable, empty **List** object of the inferred type. |
| static <K, V> Map<K, V> emptyMap( ) | Returns an immutable, empty **Map** object of the inferred type. |
| static <T> Set<T> emptySet( ) | Returns an immutable, empty **Set** object of the inferred type. |
| static <T> Enumeration<T><br>enumeration(Collection<T> *c*) | Returns an enumeration over *c*. (See "The Enumeration Interface," later in this chapter.) |
| static <T> void fill(List<? super T> *list*, T *obj*) | Assigns *obj* to each element of *list*. |

**TABLE 17-14** The Algorithms Defined by **Collections**

# Collections - The Collection Algorithm

| Method | Description |
|---|---|
| static int frequency(Collection<?> *c*, Object *obj*) | Counts the number of occurrences of *obj* in *c* and returns the result. |
| static int indexOfSubList(List<?> *list*, List<?> *subList*) | Searches *list* for the first occurrence of *subList*. Returns the index of the first match, or –1 if no match is found. |
| static int lastIndexOfSubList(List<?> *list*, List<?> *subList*) | Searches list for the last occurrence of *subList*. Returns the index of the last match, or –1 if no match is found. |
| static <T> ArrayList<T> list(Enumeration<T> *enum*) | Returns an **ArrayList** that contains the elements of *enum*. |
| static <T> T max(Collection<? extends T> *c*, Comparator<? super T> *comp*) | Returns the maximum element in *c* as determined by *comp*. |
| static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> *c*) | Returns the maximum element in *c* as determined by natural ordering. The collection need not be sorted. |
| static <T> T min(Collection<? extends T> *c*, Comparator<? super T> *comp*) | Returns the minimum element in *c* as determined by *comp*. The collection need not be sorted. |
| static <T extends Object & Comparable<? superT>> T min(Collection<? extends T> *c*) | Returns the minimum element in *c* as determined by natural ordering. |
| static <T> List<T> nCopies(int *num*, T *obj*) | Returns *num* copies of *obj* contained in an immutable list. *num* must be greater than or equal to zero. |
| static <E> Set<E> newSetFromMap(Map<E, Boolean> *m*) | Creates and returns a set backed by the map specified by *m*, which must be empty at the time this method is called. (Added by Java SE 6.) |

# Collections - The Collection Algorithm

| | |
|---|---|
| static <T> boolean replaceAll(List<T> *list*, T *old*, T *new*) | Replaces all occurrences of *old* with *new* in *list*. Returns **true** if at least one replacement occurred. Returns **false**, otherwise. |
| static void reverse(List<T> *list*) | Reverses the sequence in *list*. |
| static <T> Comparator<T> reverseOrder(Comparator<T> *comp*) | Returns a reverse comparator based on the one passed in *comp*. That is, the returned comparator reverses the outcome of a comparison that uses *comp*. |
| static <T> Comparator<T> reverseOrder( ) | Returns a reverse comparator, which is a comparator that reverses the outcome of a comparison between two elements. |
| static void rotate(List<T> *list*, int *n*) | Rotates *list* by *n* places to the right. To rotate left, use a negative value for *n*. |
| static void shuffle(List<T> *list*, Random *r*) | Shuffles (i.e., randomizes) the elements in *list* by using *r* as a source of random numbers. |
| static void shuffle(List<T> *list*) | Shuffles (i.e., randomizes) the elements in *list*. |
| static <T> Set<T> singleton(T *obj*) | Returns *obj* as an immutable set. This is an easy way to convert a single object into a set. |
| static <T> List<T> singletonList(T *obj*) | Returns *obj* as an immutable list. This is an easy way to convert a single object into a list. |
| static <K, V> Map<K, V> singletonMap(K *k*, V *v*) | Returns the key/value pair *k*/*v* as an immutable map. This is an easy way to convert a single key/value pair into a map. |

**TABLE 17-14** The Algorithms Defined by **Collections** *(continued)*

# Collections - The Collection Algorithm

| Method | Description |
|---|---|
| static <T> void sort(List<T> *list*, Comparator<? super T> *comp*) | Sorts the elements of *list* as determined by *comp*. |
| static <T extends Comparable<? super T>> void sort(List<T> *list*) | Sorts the elements of *list* as determined by their natural ordering. |
| static void swap(List<?> *list*, int *idx1*, int *idx2*) | Exchanges the elements in *list* at the indices specified by *idx1* and *idx2*. |
| static <T> Collection<T> synchronizedCollection(Collection<T> *c*) | Returns a thread-safe collection backed by *c*. |
| static <T> List<T> synchronizedList(List<T> *list*) | Returns a thread-safe list backed by *list*. |
| static <K, V> Map<K, V> synchronizedMap(Map<K, V> *m*) | Returns a thread-safe map backed by *m*. |
| static <T> Set<T> synchronizedSet(Set<T> *s*) | Returns a thread-safe set backed by *s*. |
| static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> *sm*) | Returns a thread-safe sorted map backed by *sm*. |
| static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> *ss*) | Returns a thread-safe set backed by *ss*. |
| static <T> Collection<T> unmodifiableCollection( Collection<? extends T> *c*) | Returns an unmodifiable collection backed by *c*. |
| static <T> List<T> unmodifiableList(List<? extends T> *list*) | Returns an unmodifiable list backed by *list*. |

# Collections - The Collection Algorithm

| | |
|---|---|
| static <K, V> Map<K, V><br>unmodifiableMap(Map<? extends K,<br>? extends V> *m*) | Returns an unmodifiable map backed by *m*. |
| static <T> Set<T><br>unmodifiableSet(Set<? extends T> *s*) | Returns an unmodifiable set backed by *s*. |
| static <K, V> SortedMap<K, V><br>unmodifiableSortedMap(SortedMap<K,<br>? extends V> *sm*) | Returns an unmodifiable sorted map backed by *sm*. |
| static <T> SortedSet<T><br>unmodifiableSortedSet(SortedSet<T> *ss*) | Returns an unmodifiable sorted set backed by *ss*. |

**TABLE 17-14** The Algorithms Defined by **Collections** *(continued)*

**Collections - The Collection Algorithm**

**Collections.addAll( )**

```
TreeSet<Integer> a = new TreeSet();
Collections.addAll(a, 1,2,3,4);    System.out.println(a);
```

**User-defined type**

```
class student implements Comparable<student>  {
    private String usn;
    public student(String u)  { usn = u; }

    public String toString()   {  return usn;  }

    @Override
    public int compareTo(student o) {
        int v= usn.compareTo(o.usn);
        if (v > 0) return 1;
        if (v == 0) return 0;
        return -1;       }   }
```

**Collections - The Collection Algorithm**

**User-defined value**

```
TreeSet<student> a = new TreeSet();
Collections.addAll(a, new student("1"), new student("2"), new student("3"));
System.out.println(a);
```

**Collections.asLifoQueue( )**

```
Deque<Integer> d = new LinkedList<Integer>();
d.add(3);   d.add(1);    //d is LinkedList which maintains insertion order
Queue<Integer> q = Collections.asLifoQueue(d);
q.add(9);  q.add(10);  q.add(11);  //q is a LIFO queue
System.out.println(q);                  // o/p  [11 10  9 3 1]
System.out.println(q.remove());     //11
System.out.println(q.remove());     //10
```

## Collections - The Collection Algorithm

## Collections.binarySearch( )

```
List<Integer> d = new LinkedList<Integer>();
d.add(1);   d.add(2);   d.add(3);   d.add(4);

int v = Collections.binarySearch(d, 4, null);
System.out.println(v);

v = Collections.binarySearch(d, 40, null);
System.out.println(v);
```

Output: 3

-5

**Collections - The Collection Algorithm**

**Collections.binarySearch( ) : User defined types**

```
class student implements Comparable<student>  {
        private String usn;         private int marks;
        public student(String u,int m)
        { usn = u; marks=m; }
        public String toString()
        {  return usn +" " + marks;  }
        public int getmarks() {return marks;}
        public int compareTo(student o) {
                int v = usn.compareTo(o.usn);
                if (v > 0) return 1;
                if (v == 0) return 0;
                return -1;
        }
}
```

**Collections - The Collection Algorithm**

**Collections.binarySearch( ) : User defined types**

```
class TheComparator implements Comparator<student>
{
    @Override
    public int compare(student o1, student o2) {
        if (o1.getmarks() > o2.getmarks()) return 1;
        if (o1.getmarks() == o2.getmarks()) return 0;
        return -1;
    }

}
```

## Collections - The Collection Algorithm
## Collections.binarySearch( ) : User defined types

```
class First  {
 public static void main(String[] args) {
      List<student> d = new LinkedList<student>();
      d.add(new student("1",90));   d.add(new student("2",91));
      d.add(new student("3",92));   d.add(new student("4",93));

      int v = Collections.binarySearch(d, new student("1",90), null);
      System.out.println(v);

      v = Collections.binarySearch(d, new student("4",93),new TheComparator());
      System.out.println(v);
      }
}
```

**Collections - The Collection Algorithm**

**Collections.sort(List<T> list ) : User defined types**

```
List<Integer> c = new LinkedList<Integer>();
c.add(1));
c.add(0);
c.add(4);

System.out.println(c);

Collections.sort(c);
System.out.println(c);
```

Output: [1, 0, 4]

[0, 1, 4]

**Collections - The Collection Algorithm**

**Collections.sort(List<T> list, Comparator<? super T> comp ) : User defined types**

```
class TheComparable implements Comparator<student>    {
        public int compare(student o1, student o2) {
                System.out.println("In compare func ");
                int v = (o1.toString()).compareTo(o2.toString());
                if (v > 0) return -1;
                if (v==0) return 0;
                return 1;       }           }

        List<student> c = new LinkedList();
        c.add(new student("1"));       c.add(new student("0"));
        c.add(new student("4"));              System.out.println(c);

        Collections.sort(c, new TheComparable());
        System.out.println(c);
```

**Collections - The Collection Algorithm**

**Collections.checkedList(List<E> c, Class<E> t): User defined types**

```
List<student> c = new LinkedList();
c.add(new student("1"));        c.add(new student("0"));
c.add(new student("4"));

System.out.println(c);

List<student> d = Collections.checkedList(c, student.class);

System.out.println(d);

d.add((student)new Object());  //*
System.out.println(d);
```

* Generates Class cast exception

**Collections - Why Generic Collections?**

The Collections Framework is the single most important use of generics in the Java API.

The reason for this is that generics add **type safety** to the Collections Framework.

An example that uses pre-generics code. The following program stores a list of strings in an ArrayList and then displays the contents of the list:

Ex: // **Pre-generics example** that uses a collection.

```
import java.util.*;
class Main {
  public static void main(String args[]) {
    ArrayList list = new ArrayList();
/* These lines store strings, but any type of object can be stored.  In old-style code, there is no
    convenient way to restrict the type of objects stored in a collection  */
    list.add("one");     list.add("two");     list.add("three");     list.add("four");

    list.add(12);   // this generates ClassCastException while retrieving value
```

## Collections - Why Generic Collections?

```
        Iterator itr = list.iterator();
        while(itr.hasNext()) {
/*To retrieve an element, an explicit type cast is needed because the collection stores only Object. */
            String str = (String) itr.next(); // explicit cast needed here.
            System.out.println(str + " is " + str.length() + " chars long.");
        }
} }
```

The **First** problem with pre-generics is, it stores references of type **Object**, which allowed any type of information to be stored in the collection.

The preceding program uses this feature to store references to objects of type **String** in **list**, but any type of information can be stored.

**Collections - Why Generic Collections?**

This way of storing **Object** references, leads to errors easily.

First, it is required that the programmer, rather than the compiler, ensure that only objects of the proper type be stored in a specific collection.

Ex: in the preceding program, **list** is clearly intended to store **String**s, but there is nothing that actually prevents another type of reference from being added to the collection. Compiler will not generate any error for the code below.

```
list.add(100);
```

The **Second** problem with pre-generics collections is that when a reference is retrieved from the collection, it must be manually casted into the proper type. This is why the preceding program casts the reference returned by **next( )** into **String**.

Pre-generics, collections has **Object** references. Thus, the cast is necessary when retrieving objects value from collection.

**Collections - Why Generic Collections?**

The lack of type safety often lead to errors. Because, **Object** reference can be cast into any type of object, it is possible to cast a reference obtained from a collection into the *wrong type.*

Ex: if the following statement were added to the preceding example, it would still compile without error, but generate a run-time exception when executed:

**Integer i = (Integer) itr.next();**

The addition of generics fundamentally improves the usability and safety of collections because it,

- Ensures that only references to objects of the proper type can actually be stored in a collection. Thus, a collection will always contain references of a known type.

- Eliminates the need to cast a reference retrieved from a collection.
  Instead, a reference retrieved from a collection is automatically cast into the proper type.
  This prevents run-time errors due to invalid casts and avoids an entire category of errors.

**Collections - Why Generic Collections?**

These two improvements are made possible because each collection class has been given a type parameter that specifies the type of the collection.

For example, **ArrayList** is now declared like this:

**class ArrayList<E>**

Here, **E** is the type of element stored in the collection.

Therefore, the following declares an **ArrayList** for objects of type **String**:

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

Now, only references of type **String** can be added to the list.

**Legacy classes and interfaces in Java - has to be considered**