# COLLECTIONS IN JAVA

The Collection interface defines the common operations for lists, vectors, stacks, queues, priority queues, and sets.

The Java Collections Framework supports two types of containers:

- One for storing a collection of elements is simply called a collection.
- The other, for storing key/value pairs, is called a map.

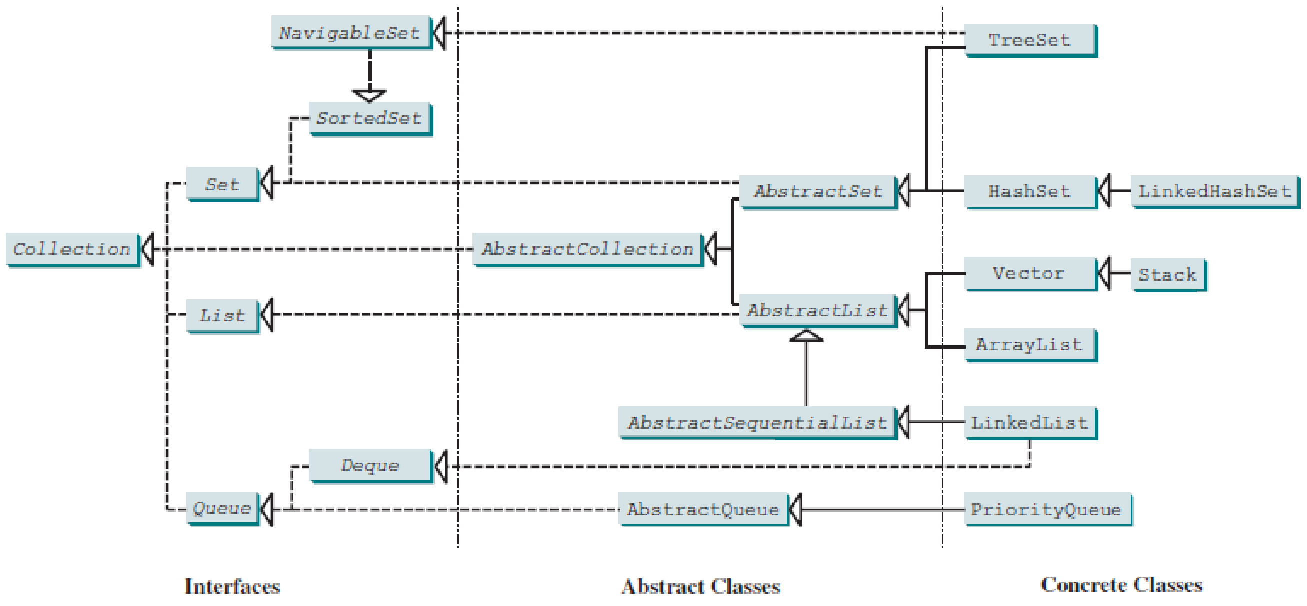
Maps are efficient data structures for quickly searching an element using a key.

- **Sets** store a group of nonduplicate elements.
- **Lists** store an ordered collection of elements.
- **Stacks** store objects that are processed in a last-in, first-out fashion.
- **Queues** store objects that are processed in a first-in, first-out fashion.
- **PriorityQueues** store objects that are processed in the order of their priorities.

The common operations of these collections are defined in the interfaces, and implementations are provided in concrete classes,

**All the interfaces and classes defined in the Java Collections Framework are grouped in the **java.util** package.**

The design of the Java Collections Framework is an excellent example of using interfaces, abstract classes, and concrete classes. The interfaces define the common operations.



The abstract classes provide partial implementation. The concrete classes implement the interfaces with concrete data structures. Providing an abstract class that partially implements an interface makes it convenient for the user to write the code. The user can simply define a concrete class that extends the abstract class rather than implementing all the methods in the interface. The abstract classes such as **AbstractCollection** are provided for convenience. For this reason, they are called convenience abstract classes.

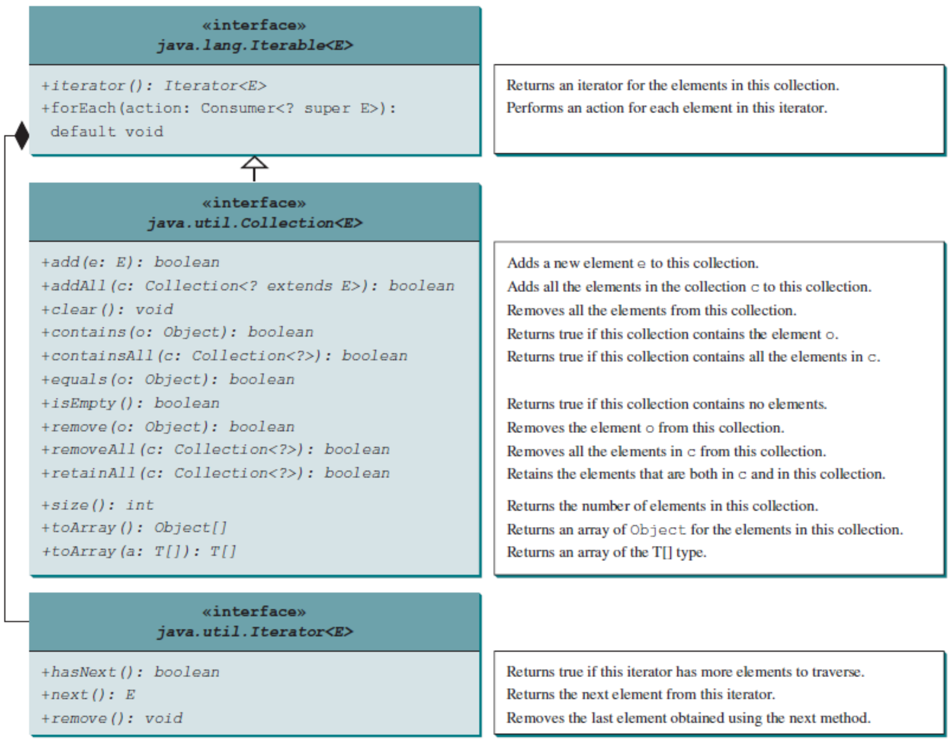
The **Collection** interface is the root interface for manipulating a collection of objects. The **AbstractCollection** class provides partial implementation for the **Collection** interface. It implements all the methods in **Collection** except the **add**, **size**, and **iterator** methods. These are implemented in the concrete subclasses.

The **Collection** interface provides the basic operations for adding and removing elements in a collection. The **add** method adds an element to the collection. The **addAll** method adds all the elements in the specified collection to this collection. The **remove** method removes an element from the collection. The **removeAll** method removes the elements from this collection that are present in the specified collection. The **retainAll** method retains the elements in this collection that are also present in the specified collection. All these methods return **boolean**. The return value is **true** if the collection is changed as a result of the method execution. The **clear()** method simply removes all the elements from the collection.

**The methods **addAll**, **removeAll**, and **retainAll** are similar to the set union, difference, and intersection operations.**

The **Collection** interface provides various query operations. The **size** method returns the number of elements in the collection. The **contains** method checks whether the collection contains the specified element. The **containsAll** method checks whether the collection contains all the elements in the specified collection. The **isEmpty** method returns **true** if the collection is empty.

The **Collection** interface provides the **toArray()** method, which returns an array of **Object** for the collection. It also provides the **toArray(T[])** method, which returns an array of the **T[]** type.


«interface»
java.lang.Iterable<E>
+iterator(): Iterator<E>
+forEach(action: Consumer<? super E>): default void
Returns an iterator for the elements in this collection.
Performs an action for each element in this iterator.
«interface»
java.util.Collection<E>
+add(e: E): boolean
+addAll(c: Collection<? extends E>): boolean
+clear(): void
+contains(o: Object): boolean
+containsAll(c: Collection<?>): boolean
+equals(o: Object): boolean
+isEmpty(): boolean
+remove(o: Object): boolean
+removeAll(c: Collection<?>): boolean
+retainAll(c: Collection<?>): boolean
+size(): int
+toArray(): Object[]
+toArray(a: T[]): T[]
Adds a new element e to this collection.
Adds all the elements in the collection c to this collection.
Removes all the elements from this collection.
Returns true if this collection contains the element o.
Returns true if this collection contains all the elements in c.
Returns true if this collection contains no elements.
Removes the element o from this collection.
Removes all the elements in c from this collection.
Retains the elements that are both in c and in this collection.
Returns the number of elements in this collection.
Returns an array of Object for the elements in this collection.
Returns an array of the T[] type.
«interface»
java.util.Iterator<E>
+hasNext(): boolean
+next(): E
+remove(): void
Returns true if this iterator has more elements to traverse.
Returns the next element from this iterator.
Removes the last element obtained using the next method.

**Some of the methods in the **Collection** interface cannot be implemented in the concrete subclass. In this case, the method would throw **java.lang.UnsupportedOperation Exception**, a subclass of **RuntimeException**.**

Example to use the methods defined in the **Collection** interface.

**`TestCollection.java`**

```java
import java.util.*;

public class TestCollection {
   public static void main(String[] args) {
         ArrayList<String> collection1 = new ArrayList<>();
         collection1.add("New York");
         collection1.add("Atlanta");
         collection1.add("Dallas");
         collection1.add("Madison");

         System.out.println("A list of cities in collection1:");
         System.out.println(collection1);

14          System.out.println("\nIs Dallas in collection1? "
15          + collection1.contains("Dallas"));
16
17          collection1.remove("Dallas");
18          System.out.println("\n" + collection1.size() +
19          " cities are in collection1 now");
20
21          Collection<String> collection2 = new ArrayList<>();
22          collection2.add("Seattle");
23          collection2.add("Portland");
24          collection2.add("Los Angeles");
25          collection2.add("Atlanta");
26
27          System.out.println("\nA list of cities in collection2:");
28          System.out.println(collection2);
29
30          ArrayList<String> c1 = (ArrayList<String>)(collection1.clone());
31          c1.addAll(collection2);
32          System.out.println("\nCities in collection1 or collection2: ");
33          System.out.println(c1);
34
35          c1 = (ArrayList<String>)(collection1.clone());
36          c1.retainAll(collection2);
37          System.out.print("\nCities in collection1 and collection2: ");
38          System.out.println(c1);
39
40          c1 = (ArrayList<String>)(collection1.clone());
41          c1.removeAll(collection2);
42          System.out.print("\nCities in collection1, but not in 2: ");
43          System.out.println(c1);
44     }
45 }
```

Output:

```
A list of cities in collection1:
[New York, Atlanta, Dallas, Madison]
Is Dallas in collection1? true
3 cities are in collection1 now
A list of cities in collection2:
[Seattle, Portland, Los Angeles, Atlanta]
Cities in collection1 or collection2:
[New York, Atlanta, Madison, Seattle, Portland, Los Angeles, Atlanta]
Cities in collection1 and collection2: [Atlanta]
Cities in collection1, but not in 2: [New York, Madison]
```

The program creates a concrete collection object using **ArrayList** (line 5) and invokes the **Collection** interface's **contains** method (line 15), **remove** method (line 17), **size** method (line 18), **addAll** method (line 31), **retainAll** method (line 36), and **removeAll** method (line 41).

For this example, we use **ArrayList**. You can use any concrete class of **Collection** such as **HashSet** and **LinkedList** to replace **ArrayList** to test these methods defined in the **Collection** interface. The program creates a copy of an array list (lines 30, 35, and 40). The purpose of this is to keep the original array list intact and use its copy to perform **addAll**, **retainAll**, and **removeAll** operations.

**All the concrete classes in the Java Collections Framework implement the **java.lang .Cloneable** and **java.io.Serializable** interfaces except that **java.util .PriorityQueue** does not implement the **Cloneable** interface. Thus, all instances of **Collection** except priority queues can be cloned and all instances of **Collection** can be serialized.**

## Iterators

*Each collection is* **Iterable**. *You can obtain its* **Iterator** *object to traverse all the elements in the collection.* **Iterator** is a classic design pattern for walking through a data structure without having to expose the details of how data is stored in the data structure.

The **Collection** interface extends the **Iterable** interface. The **Iterable** interface defines the **iterator** method, which returns an iterator. The **Iterator** interface provides a uniform way for traversing elements in various types of collections. The **iterator()** method in the **Iterable** interface returns an instance of **Iterator**, as shown in Figure 20.2, which provides sequential access to the elements in the collection using the **next()** method. You can also use the **hasNext()** method to check whether there are more elements in the iterator, and the **remove()** method to remove the last element returned by the iterator. TestIterator.java that uses the iterator to traverse all the elements in an array list.

**TestIterator.java**

```
1 import java.util.*;
2
3 public class TestIterator {
4    public static void main(String[] args) {
5          Collection<String> collection = new ArrayList<>();
6          collection.add("New York");
7          collection.add("Atlanta");
8          collection.add("Dallas");
9          collection.add("Madison");
10
11         Iterator<String> iterator = collection.iterator();
12         while (iterator.hasNext()) {
13              System.out.print(iterator.next().toUpperCase() + " ");
14         }
15         System.out.println();
16   }
17 }
```

OUTPUT:

```
NEW YORK ATLANTA DALLAS MADISON
```

The program creates a concrete collection object using **ArrayList** (line 5) and adds four strings into the list (lines 6–9). The program then obtains an iterator for the collection (line 11) and uses the iterator to traverse all the strings in the list and displays the strings in uppercase (lines 12–14).

You can simplify the code in lines 11–14 using a foreach loop without using an iterator, as follows:

```
for (String element: collection)
      System.out.print(element.toUpperCase() + " ");
```

This loop is read as "for each element in the collection, do the following." The foreach loop can be used for arrays as well as any instance of **Iterable**.

## Using the forEach Method

*You can use the* **forEach** *method to perform an action for each element in a collection.*

Java 8 added a new default method **forEach** in the **Iterable** interface. The method takes an argument for specifying the action, which is an instance of a functional interface **Consumer<? super E>**. The **Consumer** interface defines the **accept(E e)** method for performing an action on the element **e**.

**TestForEach.java**

```
import java.util.*;

public class TestForEach {
   public static void main(String[] args) {
         Collection<String> collection = new ArrayList<>();
         collection.add("New York");
         collection.add("Atlanta");
         collection.add("Dallas");
         collection.add("Madison");

         collection.forEach(e -> System.out.print(e.toUpperCase() + " "));
   }
}
```

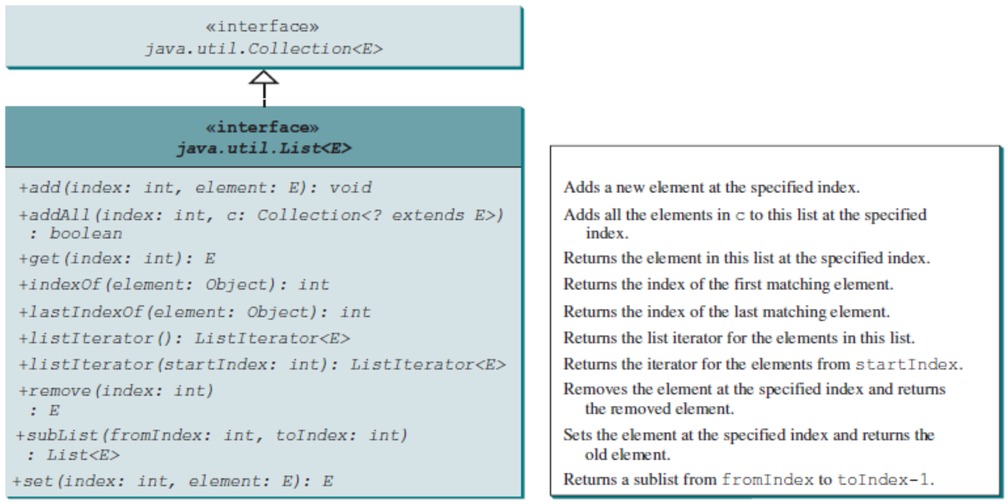
```
OUTPUT:
NEW YORK ATLANTA DALLAS MADISON
```

## LISTS

*The* **List** *interface extends the* **Collection** *interface and defines a collection for storing elements in a sequential order. To create a list, use one of its two concrete*

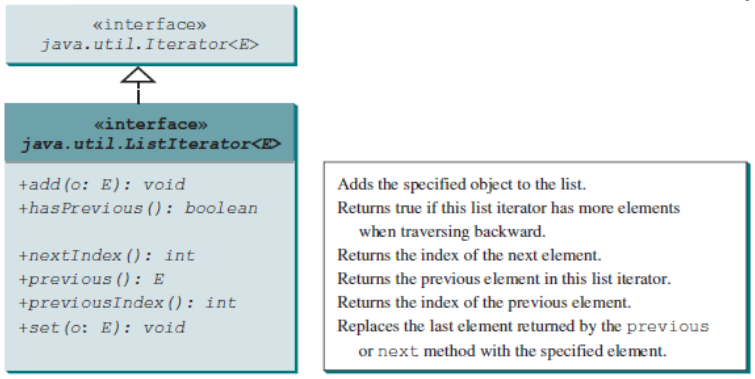
**The Common Methods in the List Interface**

**ArrayList** and **LinkedList** are defined under the **List** interface. The **List** interface extends **Collection** to define an ordered collection with duplicates allowed. The **List** interface adds position-oriented operations as well as a new list iterator that enables a list to be traversed bidirectionally


«interface»
java.util.Collection<E>
«interface»
java.util.List<E>
+add(index: int, element: E): void
+addAll(index: int, c: Collection<? extends E>) : boolean
+get(index: int): E
+indexOf(element: Object): int
+lastIndexOf(element: Object): int
+listIterator(): ListIterator<E>
+listIterator(startIndex: int): ListIterator<E>
+remove(index: int) : E
+subList(fromIndex: int, toIndex: int) : List<E>
+set(index: int, element: E): E
Adds a new element at the specified index.
Adds all the elements in c to this list at the specified index.
Returns the element in this list at the specified index.
Returns the index of the first matching element.
Returns the index of the last matching element.
Returns the list iterator for the elements in this list.
Returns the iterator for the elements from startIndex.
Removes the element at the specified index and returns the removed element.
Sets the element at the specified index and returns the old element.
Returns a sublist from fromIndex to toIndex-1.

The **add(index, element)** method is used to insert an element at a specified index and the **addAll(index, collection)** method to insert a collection of elements at a specified index. The **remove(index)** method is used to remove an element at the specified index from the list. A new element can be set at the specified index using the **set(index, element)** method.

The **indexOf(element)** method is used to obtain the index of the specified element's first occurrence in the list and the **lastIndexOf(element)** method to obtain the index of its last occurrence. A sublist can be obtained by using the **subList(fromIndex, toIndex)** method. The **listIterator()** or **listIterator(startIndex)** method returns an instance


«interface»
java.util.Iterator<E>
«interface»
java.util.ListIterator<E>
+add(o: E): void
+hasPrevious(): boolean
+nextIndex(): int
+previous(): E
+previousIndex(): int
+set(o: E): void
Adds the specified object to the list.
Returns true if this list iterator has more elements when traversing backward.
Returns the index of the next element.
Returns the previous element in this list iterator.
Returns the index of the previous element.
Replaces the last element returned by the previous or next method with the specified element.

of **ListIterator**. The **ListIterator** interface extends the **Iterator** interface to add bidirectional traversal of the list.

The **add(element)** method inserts the specified element into the list. The element is inserted immediately before the next element that would be returned by the **next()** method defined in the **Iterator** interface, if any, and after the element that would be returned by the **previous()** method, if any. If the list doesn't contain any elements, the new element becomes the sole element in the list. The **set(element)** method can be used to replace the last element returned by the **next** method, or the **previous** method with the specified element.

The **hasNext()** method defined in the **Iterator** interface is used to check whether the iterator has more elements when traversed in the forward direction, and the **hasPrevious()** method to check whether the iterator has more elements when traversed in the backward direction.

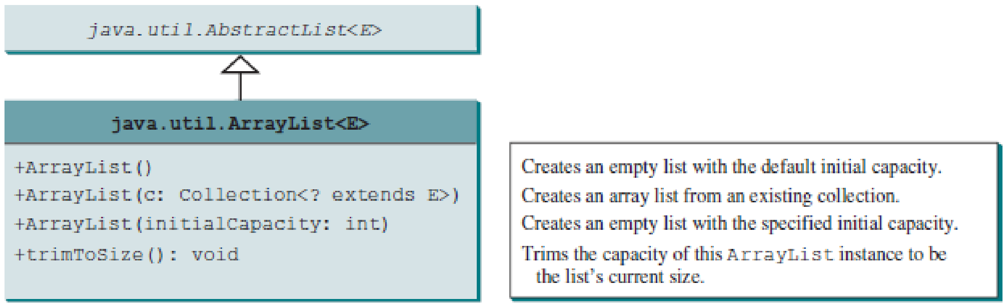
The **next()** method defined in the **Iterator** interface returns the next element in the iterator, and the **previous()** method returns the previous element in the iterator. The **nextIndex()** method returns the index of the next element in the iterator, and the **previousIndex()** returns the index of the previous element in the iterator.

The **AbstractList** class provides a partial implementation for the **List** interface. The **AbstractSequentialList** class extends **AbstractList** to provide support for linked lists.

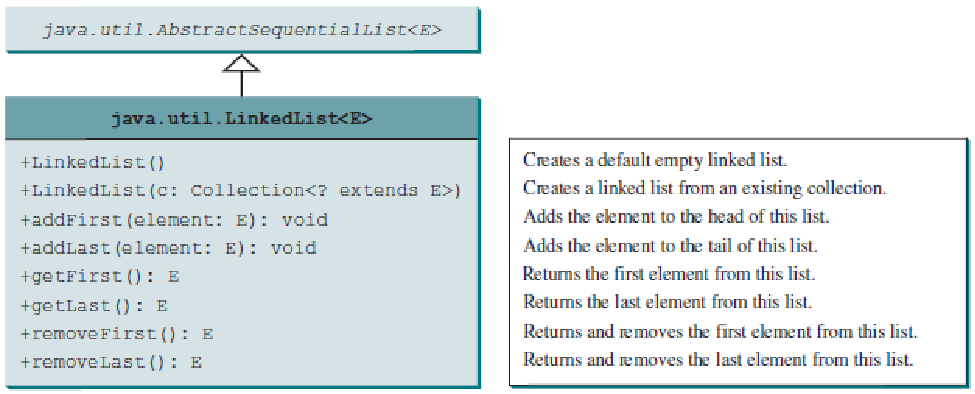
The **ArrayList** and **LinkedList** Classes

The **ArrayList** class and the **LinkedList** class are two concrete implementations of the **List** interface. **ArrayList** stores elements in an array. The array is dynamically created. If the capacity of the array is exceeded, a larger new array is created and all the elements from the current array are copied to the new array. **LinkedList** stores elements in a *linked list*. Which of the two classes you use depends on your specific needs. If you need to support random access through an index without inserting or removing elements at the beginning of the list, **Array-List** is the most efficient. If, however, your application requires the insertion or deletion of elements at the beginning of the list, you should choose **LinkedList**. A list can grow or shrink dynamically. Once it is created, an array is fixed. If your application does not require the insertion or deletion of elements, an array is the most efficient data structure.

**ArrayList** is a resizable-array implementation of the **List** interface. It also provides methods for manipulating the size of the array used internally to store the list. Each **ArrayList** instance has a capacity, which is the size of the array used tostore the elements in the list. It is always at least as large as the list size. As elements are added to an **ArrayList**, its capacity grows automatically. An **ArrayList** does not automatically shrink. You can use the **trimToSize()** method to reduce the array capacity to the size of the list. An **ArrayList** can be constructed using its no-arg constructor, **ArrayList(Collection)**, or **ArrayList(initialCapacity)**.


java.util.AbstractList<E>
java.util.ArrayList<E>
+ArrayList()
+ArrayList(c: Collection<? extends E>)
+ArrayList(initialCapacity: int)
+trimToSize(): void
Creates an empty list with the default initial capacity.
Creates an array list from an existing collection.
Creates an empty list with the specified initial capacity.
Trims the capacity of this ArrayList instance to be the list's current size.

**LinkedList** is a linked list implementation of the **List** interface. In addition to implementing the **List** interface, this class provides the methods for retrieving, inserting, and removing elements from both ends of the list. A **LinkedList** can be constructed using its no-arg constructor or **LinkedList(Collection)**.


java.util.AbstractSequentialList<E>
java.util.LinkedList<E>
+LinkedList()
+LinkedList(c: Collection<? extends E>)
+addFirst(element: E): void
+addLast(element: E): void
+getFirst(): E
+getLast(): E
+removeFirst(): E
+removeLast(): E
Creates a default empty linked list.
Creates a linked list from an existing collection.
Adds the element to the head of this list.
Adds the element to the tail of this list.
Returns the first element from this list.
Returns the last element from this list.
Returns and removes the first element from this list.
Returns and removes the last element from this list.

The next program that creates an array list filled with numbers and inserts new elements into specified locations in the list. The example also creates a linked list from the array list and inserts and removes elements from the list. Finally, the example traverses the list forward and backward.

**TestArrayAndLinkedList.java**

```
import java.util.*;

public class TestArrayAndLinkedList {
        public static void main(String[] args) {
                List<Integer> arrayList = new ArrayList<>();
                arrayList.add(1); // 1 is autoboxed to an Integer object
                arrayList.add(2);
                arrayList.add(3);
                arrayList.add(1);
                arrayList.add(4);
                arrayList.add(0, 10);
                arrayList.add(3, 30);

                System.out.println("A list of integers in the array list:");
                System.out.println(arrayList);

                LinkedList<Object> linkedList = new LinkedList<Object>(arrayList);
                linkedList.add(1, "red");
                linkedList.removeLast();
                linkedList.addFirst("green");

                System.out.println("Display the linked list forward:");
                ListIterator<Object> listIterator = linkedList.listIterator();
                while (listIterator.hasNext()) {
                    System.out.print(listIterator.next() + " ");
                }
                System.out.println();

                System.out.println("Display the linked list backward:");
                listIterator = linkedList.listIterator(linkedList.size());
                while (listIterator.hasPrevious()) {
                    System.out.print(listIterator.previous() + " ");
                }
        }
}
```

**OUTPUT:**

```
A list of integers in the array list:
[10, 1, 2, 30, 3, 1, 4]
Display the linked list forward:
green 10 red 1 2 30 3 1
```

```
Display the linked list backward:
1 3 30 2 1 red 10 green
```

A list can hold identical elements. Integer **1** is stored twice in the list (lines 6 and 9). **ArrayList** and **LinkedList** operate similarly. The critical difference between them pertains to internal implementation, which affects their performance. **LinkedList** is efficient for inserting and removing elements at the beginning of the list, and **ArrayList** is more efficient for all other operations.

The **get(i)** method is available for a linked list, but it is a time-consuming operation. Do not use it to traverse all the elements in a list as shown in (a). Instead, you should use a foreach loop as shown in (b) or a **forEach** method as shown in (c). Note (b) and (c) use an iterator implicitly.

```
for (int i = 0; i < list.size(); i++)
  process list.get(i);
}
```

(a) Very inefficient

```
for (listElementType e: list) {
  process e;
}
```

(b) Efficient

```
list.forEach(e ->
  process e
)
```

(c) Efficient

Java provides the static **asList** method for creating a list from a variable-length list of arguments. Thus, you can use the following code to create a list of strings and a list of integers:

```
List<String> list1 = Arrays.asList("red", "green", "blue");
List<Integer> list2 = Arrays.asList(10, 20, 30, 40, 50);
```

In Java 9, you can use the static **List.of** method to replace Arrays.**asList** for creating a list from a variable-length list of arguments. The name **List.of** is more intuitive and easier to memorize than **Arrays.asList**.

## THE **COMPARATOR** INTERFACE

**Comparator** *can be used to compare the objects of a class that doesn't implement*
**Comparable** *or define a new criteria for comparing objects.*
Several classes in the Java API, such as **String**, **Date**, **Calendar**, **BigInteger**, **BigDecimal**, and all the numeric wrapper classes for the primitive types, implement the **Comparable**interface. The **Comparable** interface defines the **compareTo** method, which is usedto compare two elements of the same class that implements the **Comparable** interface.

What if the elements' classes do not implement the **Comparable** interface? Can theseelements be compared? You can define a *comparator* to compare the elements of different classes. To do so, define a class that implements the **java.util.Comparator<T>** interface and overrides its **compare(a, b)** method.

```
public int compare(T a, T b)
Returns a negative value if a is less than b, a positive value if a is
greater than b, and zero if they are equal.
```

**Example**

The **GeometricObject** class was introduced in Section 13.2, Abstract Classes. The **GeometricObject** class does not implement the **Comparable** interface. To compare the objects of the **GeometricObject** class, you can define a comparator class.

```
GeometricObjectComparator.java
import java.util.Comparator;

public class GeometricObjectComparator
implements Comparator<GeometricObject>, java.io.Serializable {
public int compare(GeometricObject o1, GeometricObject o2) {
         double area1 = o1.getArea();
         double area2 = o2.getArea();

         if (area1 < area2)
10              return -1;
11          else if (area1 == area2)
12              return 0;
13          else
14          return 1;
15          }
16 }
```

Line 4 implements **Comparator<GeometricObject>**. Line 5 overrides the **compare** method to compare two geometric objects. The class also implements **Serializable**. It is generally a good idea for comparators to implement **Serializable** so they can be serialized. The **compare(o1,o2)** method returns **1** if **o1.getArea() > o2.getArea(),0** if **o1.getArea() == o2.getArea()**, and **-1** otherwise.

TestComparator.java gives a method that returns a larger object between two geometric objects. The objects are compared using the **GeometricObjectComparator**.

```
TestComparator.java
1 import java.util.Comparator;
2
3 public class TestComparator {
4           public static void main(String[] args) {
5               GeometricObject g1 = new Rectangle(5, 5);
6               GeometricObject g2 = new Circle(5);
7
8               GeometricObject g =
9               max(g1, g2, new GeometricObjectComparator());
10
11              System.out.println("The area of the larger object is " +
12              g.getArea());
13          }
14
15          public static GeometricObject max(GeometricObject g1,
16              GeometricObject g2, Comparator<GeometricObject> c) {
17              if (c.compare(g1, g2) > 0)
18                  return g1;
19              else
20                  return g2;
21          }
22 }
OUTPUT:
The area of the larger object is 78.53981633974483
```

The program creates a **Rectangle** and a **Circle** object in lines 5 and 6 (the **Rectangle** and **Circle** classes were defined in Section 13.2, Abstract Classes). They are all subclasses of **GeometricObject**. The program invokes the **max** method to obtain the geometric object with the larger area (lines 8 and 9).

The comparator is an object of the **Comparator** type whose **compare(a,b)** method is used to compare the two elements. The **GeometricObjectComparator** is created and passed to the **max** method (line 9) and this comparator is used in the **max** method to compare the geometric objects in line 17.

Since the **Comparator** interface is a single abstract method interface, you can use a lambda expression to simplify the code by replacing line 9 with the following code

```
max(g1, g2, (o1, o2) -> o1.getArea() > o2.getArea() ? 1 : o1.getArea() ==
o2.getArea() ? 0 : -1);
```

Here, **o1** and **o2** are two parameters in the **compare** method in the **Comparator** interface. The method returns **1** if **o1.getArea() > o2.getArea()**, **0** if **o1.getArea() == o2.getArea()**, and **−1** otherwise.

Comparing elements using the **Comparable** interface is referred to as comparing using *natural order*, and comparing elements using the **Comparator** interface is referred toas comparing using *comparator*.

The preceding example defines a comparator for comparing two geometric objects since the **GeometricObject** class does not implement the **Comparable** interface. Sometimes a class implements the **Comparable** interface, but if you would like to compare their objects using a different criteria, you can define a custom comparator.

SortStringByLength.java gives an example that compares string by their length.

```
SortStringByLength.java
1 public class SortStringByLength {
2          public static void main(String[] args) {
3          String[] cities = {"Atlanta", "Savannah", "New York", "Dallas"};
4          java.util.Arrays.sort(cities, new MyComparator());
5
6          for (String s : cities) {
7               System.out.print(s + " ");
8          }
9 }
10
11 public static class MyComparator implements{
12         java.util.Comparator<String> {
13         @Override
14         public int compare(String s1, String s2) {
15              return s1.length() - s2.length();
16         }
17 }
18 }
OUTPUT:
Dallas Atlanta Savannah New York
```

The program defines a comparator class by implementing the **Comparator** interface (lines 11 and 12). The **compare** method is implemented to compare two strings by their lengths (lines 14–16). The program invokes the **sort** method to sort an array of strings using a comparator (line 4).

Since **Comparator** is a functional interface, the code can be simplified using a lambda expression as follows:

```
java.util.Arrays.sort(cities,(s1, s2) -> {return s1.length() - s2.length();});
```

or simply

```
java.util.Arrays.sort(cities, (s1, s2) -> s1.length() - s2.length());
```

The **List** interface defines the **sort(comparator)** method that can be used to sort the elements in a list using a specified comparator. SortStringIgnoreCase.java gives an example of using a comparator to sort strings in a list by ignoring cases.

```
SortStringIgnoreCase.java
public class SortStringIgnoreCase {
         public static void main(String[] args) {
              java.util.List<String> cities = java.util.Arrays.asList
              ("Atlanta", "Savannah", "New York", "Dallas");
              cities.sort((s1, s2) -> s1.compareToIgnoreCase(s2));

              for (String s: cities) {
                   System.out.print(s + " ");
              }
         }
}
```

```
Atlanta dallas new York Savannah
```

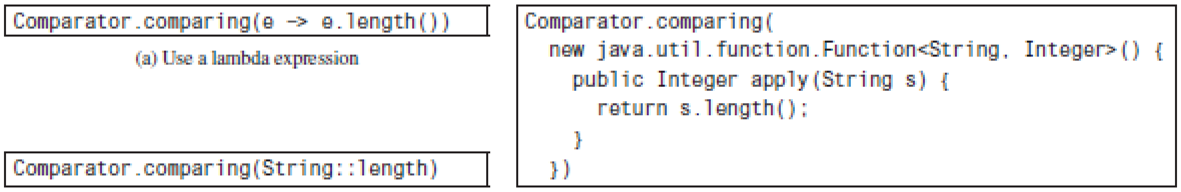
The program sorts a list of strings using a comparator that compares strings ignoring case(line 5). If you invoke **list.sort(null)**, the list will be sorted using its natural order.

The comparator is created using a lambda expression. Note the lambda expression here does nothing but simply invokes the **compareToIgnoreCase** method. In the case like this, you can use a simpler and clearer syntax to replace the lambda expression as follows:

```
cities.sort(String::compareToIgnoreCase);
```

Here **String::compareToIgnoreCase** is known as *method reference*, which is equivalent to a lambda expression. The compiler automatically translates a method reference to an equivalent lambda expression.

The **Comparator** interface also contains several useful static methods and default methods. You can use the static **comparing(Function<? sup T, ? sup R> keyExtracter)** method to create a **Comparator<T>** that compares the elements using the key extracted from a **Function** object. The **Function** object's **apply(T)** method returns the key of type **R** for the object **T**. For example, the following code in (a) creates a **Comparator** that compares strings by their length using a lambda expression, which is equivalent to the code using an anonymous inner class in (b) and a method reference in (c).



```
Comparator.comparing(e -> e.length())
```

(a) Use a lambda expression

```
Comparator.comparing(String::length)
```

```
Comparator.comparing(
  new java.util.function.Function<String, Integer>() {
    public Integer apply(String s) {
      return s.length();
    }
  })
```

The **comparing** method in the **Comparator** interface is implemented essentially as follows for the preceding example:

```
// comparing returns a Comparator
public static Comparator<String> comparing(Function<String, Integer> f) {
return (s1, s2) -> f.apply(s1).compareTo(f.apply(s2));
}
```

You can replace the comparator in Listing 20.7 using the following code:

```
java.util.Arrays.sort(cities, Comparator.comparing(String::length));
```

The **Comparator.comparing** method is particularly useful to create a **Comparator** using a property from an object. For example, the following code sorts a list of **Loan** objects, based on their **loanAmount** property.

```
Loan[] list = {new Loan(5.5, 10, 2323), new Loan(5, 10, 1000)};
Arrays.sort(list, Comparator.comparing(Loan::getLoanAmount));
```

You can sort using a primary criteria, second, third, and so on using the **Comparator**'s default **thenComparing** method. For example, the following code sorts a list of **Loan** objects first on their **loanAmount** then on **annualInterestRate**.

```
Loan[] list = {new Loan(5.5, 10, 100), new Loan(5, 10, 1000)};
Arrays.sort(list, Comparator.comparing(Loan::getLoanAmount)
.thenComparing(Loan::getAnnualInterestRate));
```

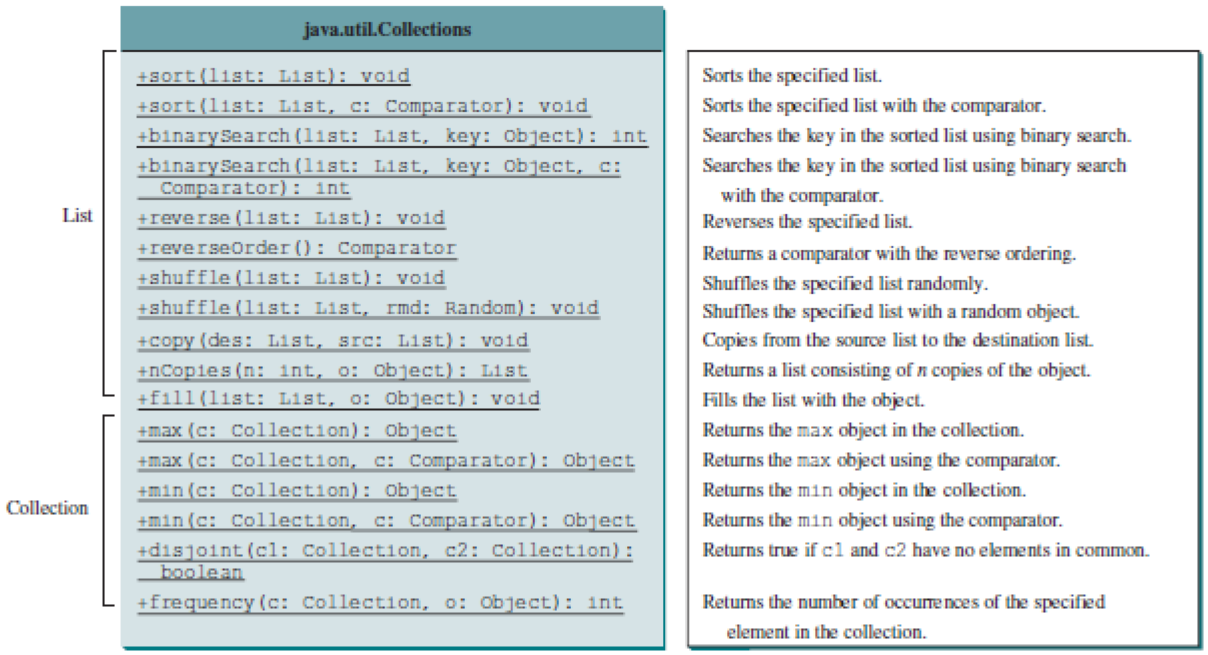
The default **reversed()** method can be used to reverse the order for a comparator. For example, the following code sorts a list of **Loan** objects on their **loanAmount** property in a decreasing order.

```
Arrays.sort(list, Comparator.comparing(Loan::getLoanAmount).
reversed());
```

## Static Methods for Lists and Collections

*The* **Collections** *class contains static methods to perform common operations in a collection and a list.*

The **Collections** class contains the **sort**, **binarySearch**, **reverse**, **shuffle**, **copy**, and **fill** methods for lists and **max**, **min**, **disjoint**, and **frequency** methods for collections,



| | java.util.Collections | |
|---|---|---|
| List | +sort(list: List): void | Sorts the specified list. |
| | +sort(list: List, c: Comparator): void | Sorts the specified list with the comparator. |
| | +binarySearch(list: List, key: Object): int | Searches the key in the sorted list using binary search. |
| | +binarySearch(list: List, key: Object, c: Comparator): int | Searches the key in the sorted list using binary search with the comparator. |
| | +reverse(list: List): void | Reverses the specified list. |
| | +reverseOrder(): Comparator | Returns a comparator with the reverse ordering. |
| | +shuffle(list: List): void | Shuffles the specified list randomly. |
| | +shuffle(list: List, rmd: Random): void | Shuffles the specified list with a random object. |
| | +copy(des: List, src: List): void | Copies from the source list to the destination list. |
| | +nCopies(n: int, o: Object): List | Returns a list consisting of *n* copies of the object. |
| | +fill(list: List, o: Object): void | Fills the list with the object. |
| Collection | +max(c: Collection): Object | Returns the max object in the collection. |
| | +max(c: Collection, c: Comparator): Object | Returns the max object using the comparator. |
| | +min(c: Collection): Object | Returns the min object in the collection. |
| | +min(c: Collection, c: Comparator): Object | Returns the min object using the comparator. |
| | +disjoint(c1: Collection, c2: Collection): boolean | Returns true if c1 and c2 have no elements in common. |
| | +frequency(c: Collection, o: Object): int | Returns the number of occurrences of the specified element in the collection. |

You can sort the comparable elements in a list in its natural order with the **compareTo** method in the **Comparable** interface. You may also specify a comparator to sort elements. For example, the following code sorts strings in a list:

```
List<String> list = Arrays.asList("red", "green", "blue");
Collections.sort(list);
System.out.println(list);
```
The output is **[blue, green, red]**.

The preceding code sorts a list in ascending order. To sort it in descending order, you can simply use the **Collections.reverseOrder()** method to return a **Comparator** object that orders the elements in reverse of natural order. For example, the following code sorts a list of strings in descending order:

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.sort(list, Collections.reverseOrder());
System.out.println(list);
```
The output is **[yellow, red, green, blue]**.

You can use the **binarySearch** method to search for a key in a list. To use this method, the list must be sorted in increasing order. If the key is not in the list, the method returns - (*insertion point* + 1). Recall that the insertion point is where the item would fall in the list if it were present. For example, the following code searches the keys in a list of integers and a sorted list of strings:

```
List<Integer> list1 =
Arrays.asList(2, 4, 7, 10, 11, 45, 50, 59, 60, 66);
System.out.println("(1) Index: " + Collections.binarySearch(list1, 7));
System.out.println("(2) Index: " + Collections.binarySearch(list1, 9));
List<String> list2 = Arrays.asList("blue", "green", "red");
System.out.println("(3) Index: " +
Collections.binarySearch(list2, "red"));
System.out.println("(4) Index: " +
Collections.binarySearch(list2, "cyan"));
```
The output of the preceding code is:

```
(1) Index: 2
(2) Index: -4
(3) Index: 2
(4) Index: -2
```
You can use the **reverse** method to reverse the elements in a list. For example, the following code displays **[blue, green, red, yellow]**:

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.reverse(list);
System.out.println(list);
```
You can use the **shuffle(List)** method to randomly reorder the elements in a list. For example, the following code shuffles the elements in **list**:

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.shuffle(list);
System.out.println(list);
```
You can also use the **shuffle(List, Random)** method to randomly reorder the elements in a list with a specified **Random** object. Using a specified **Random** object is useful to generate a list with identical sequences of elements for the same original list. For example, the following code shuffles the elements in **list**:

```
List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
List<String> list2 = Arrays.asList("yellow", "red", "green", "blue");
Collections.shuffle(list1, new Random(20));
Collections.shuffle(list2, new Random(20));
System.out.println(list1);
System.out.println(list2);
```
You will see that **list1** and **list2** have the same sequence of elements before and after the shuffling.

You can use the **copy(dest, src)** method to copy all the elements from a source list to a destination list on the same index. The destination list must be as long as the source list. If it is longer, the remaining elements in the destination list are not affected. For example, the following code copies **list2** to **list1**:

```
List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
List<String> list2 = Arrays.asList("white", "black");
Collections.copy(list1, list2);
System.out.println(list1);
```
The output for **list1** is **[white, black, green, blue]**. The **copy** method performs a shallow copy: only the references of the elements from the source list are copied.

You can use the **nCopies(int n, Object o)** method to create an immutable list that consists of **n** copies of the specified object. For example, the following code creates a list with five **Calendar** objects:

```
List<GregorianCalendar> list1 = Collections.nCopies
(5, new GregorianCalendar(2005, 0, 1));
```
The list created from the **nCopies** method is immutable, so you cannot add, remove, or update elements in the list. All the elements have the same references.

You can use the **fill(List list, Object o)** method to replace all the elements in the list with the specified element. For example, the following code displays **[black, black, black]**:

```
List<String> list = Arrays.asList("red", "green", "blue");
Collections.fill(list, "black");
System.out.println(list);
```
You can use the **max** and **min** methods for finding the maximum and minimum elements in a collection. The elements must be comparable using the **Comparable** interface or the **Comparator** interface. See the following code for examples:

```
Collection<String> collection = Arrays.asList("red", "green", "blue");
System.out.println(Collections.max(collection)); // Use Comparable
System.out.println(Collections.min(collection,
Comparator.comparing(String::length))); // Use Comparator
```

The **disjoint(collection1, collection2)** method returns **true** if the two collections have no elements in common. For example, in the following code, **disjoint(collection1, collection2)** returns **false**, but **disjoint(collection1, collection3)** returns **true**:

```
Collection<String> collection1 = Arrays.asList("red", "cyan");
Collection<String> collection2 = Arrays.asList("red", "blue");
Collection<String> collection3 = Arrays.asList("pink", "tan");
System.out.println(Collections.disjoint(collection1, collection2));
System.out.println(Collections.disjoint(collection1, collection3));
```

The **frequency(collection, element)** method finds the number of occurrences of the element in the collection. For example, **frequency(collection, "red")** returns **2** in the following code:

```
Collection<String> collection = Arrays.asList("red", "cyan", "red");
System.out.println(Collections.frequency(collection, "red"));
```

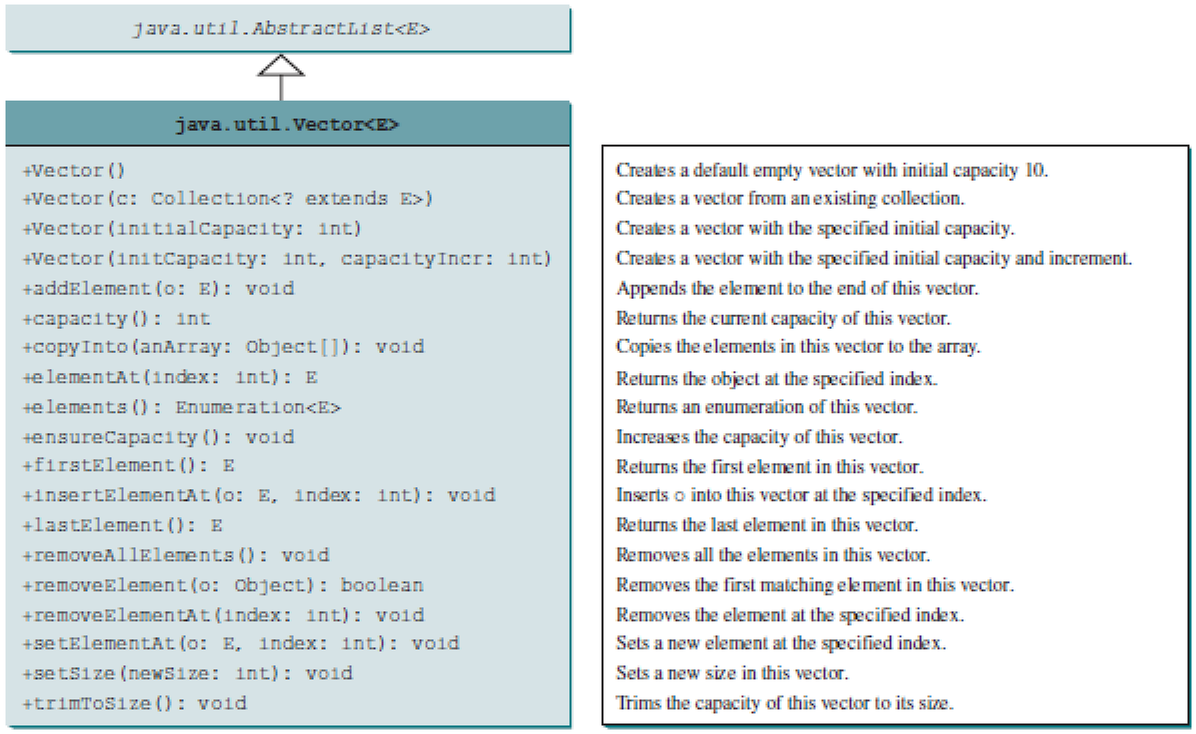
## Vector and Stack Classes

**Vector** *is a subclass of* **AbstractList** *and* **Stack** *is a subclass of* **Vector** *in the Java API.*

The Java Collections Framework was introduced in Java 2. Several data structures were supported earlier, among them the **Vector** and **Stack** classes. These classes were redesigned to fit into the Java Collections Framework, but all their old-style methods are retained for compatibility.

**Vector** is the same as **ArrayList**, except that it contains synchronized methods for accessing and modifying the vector. Synchronized methods can prevent data corruption when a vector is accessed and modified by two or more threads concurrently. For the many applications that do not require synchronization, using **ArrayList** is more efficient than using **Vector**.

The **Vector** class extends the **AbstractList** class. It also has the methods contained in the original **Vector** class defined prior to Java 2



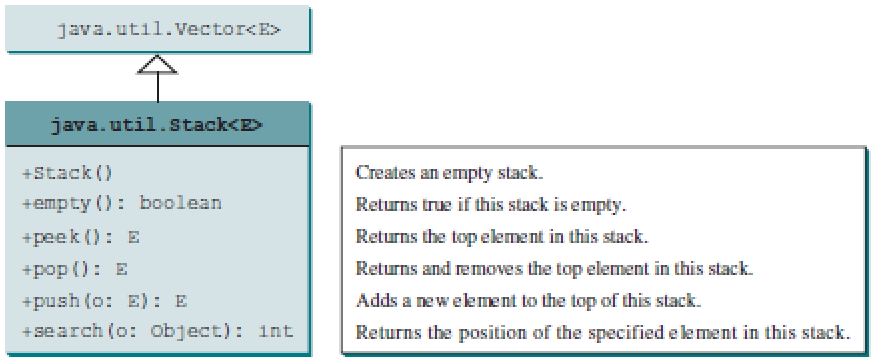
| | |
|---|---|
| +Vector() | Creates a default empty vector with initial capacity 10. |
| +Vector(c: Collection<? extends E>) | Creates a vector from an existing collection. |
| +Vector(initialCapacity: int) | Creates a vector with the specified initial capacity. |
| +Vector(initCapacity: int, capacityIncr: int) | Creates a vector with the specified initial capacity and increment. |
| +addElement(o: E): void | Appends the element to the end of this vector. |
| +capacity(): int | Returns the current capacity of this vector. |
| +copyInto(anArray: Object[]): void | Copies the elements in this vector to the array. |
| +elementAt(index: int): E | Returns the object at the specified index. |
| +elements(): Enumeration<E> | Returns an enumeration of this vector. |
| +ensureCapacity(): void | Increases the capacity of this vector. |
| +firstElement(): E | Returns the first element in this vector. |
| +insertElementAt(o: E, index: int): void | Inserts o into this vector at the specified index. |
| +lastElement(): E | Returns the last element in this vector. |
| +removeAllElements(): void | Removes all the elements in this vector. |
| +removeElement(o: Object): boolean | Removes the first matching element in this vector. |
| +removeElementAt(index: int): void | Removes the element at the specified index. |
| +setElementAt(o: E, index: int): void | Sets a new element at the specified index. |
| +setSize(newSize: int): void | Sets a new size in this vector. |
| +trimToSize(): void | Trims the capacity of this vector to its size. |

Most of the methods in the **Vector** class listed in the UML diagram are similar to the methods in the **List** interface. These methods were introduced before the Java Collections Framework. For example, **addElement(Object element)** is the same as the **add(Object element)** method, except that the **addElement** method is synchronized. Use the **ArrayList** class if you don't need synchronization. It works much faster than **Vector**.

The **elements()** method returns an **Enumeration**. The **Enumeration** interface was introduced prior to Java 2 and was superseded by the **Iterator** interface.

**Vector** is widely used in Java legacy code because it was the Java resizable-array implementation before Java 2.

In the Java Collections Framework, **Stack** is implemented as an extension of **Vector**.


java.util.Vector<E>
java.util.Stack<E>
+Stack()
+empty(): boolean
+peek(): E
+pop(): E
+push(o: E): E
+search(o: Object): int
Creates an empty stack.
Returns true if this stack is empty.
Returns the top element in this stack.
Returns and removes the top element in this stack.
Adds a new element to the top of this stack.
Returns the position of the specified element in this stack.

The **Stack** class was introduced prior to Java 2. The methods shown in above were used before Java 2. The **empty()** method is the same as **isEmpty()**. The **peek()** method looks at the element at the top of the stack without removing it. The **pop()** method removes the top element from the stack and returns it. The **push(Object element)** method adds the specified element to the stack. The **search(Object element)** method checks whether the specified element is in the stack.

## QUEUES AND PRIORITY QUEUES

*In a priority queue, the element with the highest priority is removed first.*

A *queue* is a first-in, first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue. In a *priority queue*, elements are assigned priorities. When accessing elements, the element with the highest priority is removed first.

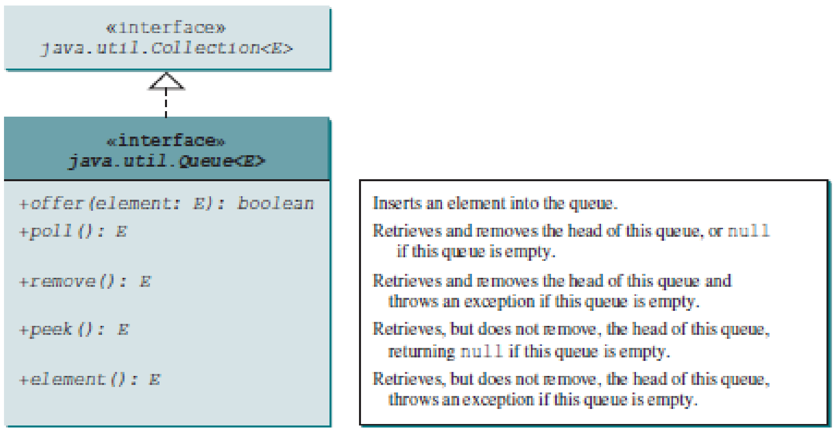
The **Queue** Interface

The **Queue** interface extends **java.util.Collection** with additional insertion, extraction, and inspection operations.

The **offer** method is used to add an element to the queue. This method is similar to the **add** method in the **Collection** interface, but the **offer** method is preferred for queues. The **poll** and **remove** methods are similar, except that **poll()** returns **null** if the queue is empty, whereas **remove()** throws an exception. The **peek** and **element** methods are similar, except that **peek()** returns **null** if the queue is empty, whereas **element()** throws an exception.

**Deque** and **LinkedList**

The **LinkedList** class implements the **Deque** interface, which extends the **Queue** interface, as shown in Figure 20.13. Therefore, you can use **LinkedList** to create a queue. **LinkedList** is ideal for queue operations because it is efficient for inserting and removing elements from both ends of a list.



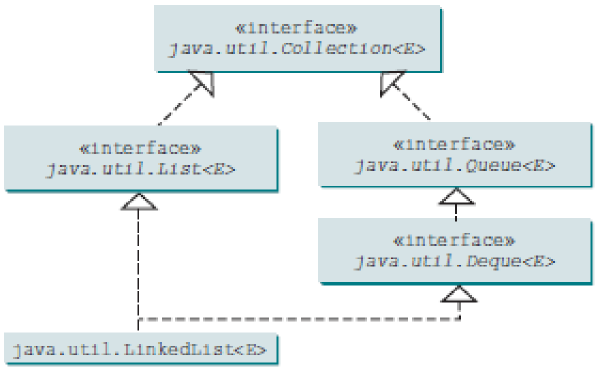
Inserts an element into the queue.

Retrieves and removes the head of this queue, or null if this queue is empty.

Retrieves and removes the head of this queue and throws an exception if this queue is empty.

Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.

Retrieves, but does not remove, the head of this queue, throws an exception if this queue is empty.

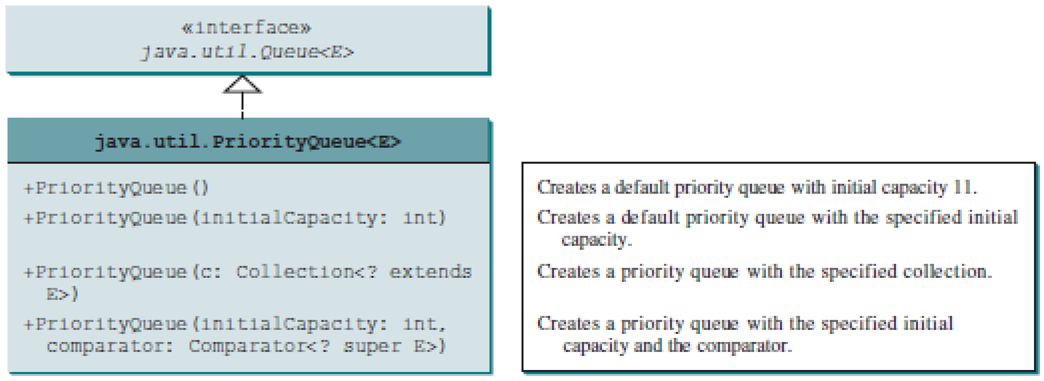


**Deque** supports element insertion and removal at both ends. The name *deque* is short for "double-ended queue" and is usually pronounced "deck." The **Deque** interface extends **Queue** with additional methods for inserting and removing elements from both ends of the queue. The methods **addFirst(e)**, **removeFirst()**, **addLast(e)**, **removeLast()**, **getFirst()**, and **getLast()** are defined in the **Deque** interface.

TestQueue.java shows an example of using a queue to store strings. Line 3 creates a queue using **LinkedList**. Four strings are added to the queue in lines 4–7. The **size()** method defined in the **Collection** interface returns the number of elements in the queue (line 9). The **remove()** method retrieves and removes the element at the head of the queue (line 10).

```
TestQueue.java
1 public class TestQueue {
2           public static void main(String[] args) {
3                java.util.Queue<String> queue = new java.util.LinkedList<>();
4                queue.offer("Oklahoma");
5                queue.offer("Indiana");
6                queue.offer("Georgia");
7                queue.offer("Texas");
8
9                while (queue.size() > 0)
10                    System.out.print(queue.remove() + " ");
11               }
12 }
OUTPUT:
Oklahoma Indiana Georgia Texas
```

The **PriorityQueue** class implements a priority queue, as shown in Figure 20.14. By default, the priority queue orders its elements according to their natural ordering using **Comparable**. The element with the least value is assigned the highest priority, and thus is removed from the queue first. If there are several elements with the same highest priority, the tie is broken arbitrarily. You can also specify an ordering using **Comparator** in the constructor **PriorityQueue(initialCapacity, comparator)**.



+PriorityQueue() — Creates a default priority queue with initial capacity 11.
+PriorityQueue(initialCapacity: int) — Creates a default priority queue with the specified initial capacity.
+PriorityQueue(c: Collection<? extends E>) — Creates a priority queue with the specified collection.
+PriorityQueue(initialCapacity: int, comparator: Comparator<? super E>) — Creates a priority queue with the specified initial capacity and the comparator.

PriorityQueueDemo.java shows an example of using a priority queue to store strings. Line 5 creates a priority queue for strings using its no-arg constructor. This priority queue orders the strings using their natural order, so the strings are removed from the queue in increasing order.

Lines 16 and 17 create a priority queue using the comparator obtained from **Collections.reverseOrder()**, which orders the elements in reverse order, so the strings are removed from the queue in decreasing order.

```
PriorityQueueDemo.java
1 import java.util.*;
2
3 public class PriorityQueueDemo {
4          public static void main(String[] args) {
5               PriorityQueue<String> queue1 = new PriorityQueue<>();
6               queue1.offer("Oklahoma");
7               queue1.offer("Indiana");
8               queue1.offer("Georgia");
9               queue1.offer("Texas");
10
11              System.out.println("Priority queue using Comparable:");
12              while (queue1.size() > 0) {
13                   System.out.print(queue1.remove() + " ");
14              }
15
16              PriorityQueue<String> queue2 = new PriorityQueue<>(
17              4, Collections.reverseOrder());
18              queue2.offer("Oklahoma");
19              queue2.offer("Indiana");
20              queue2.offer("Georgia");
21              queue2.offer("Texas");
22
23              System.out.println("\nPriority queue using Comparator:");
24              while (queue2.size() > 0) {
25                   System.out.print(queue2.remove() + " ");
26              }
27         }
28 }
```

```
OUTPUT:
Priority queue using Comparable:
Georgia Indiana Oklahoma Texas
Priority queue using Comparator:
Texas Oklahoma Indiana Georgia
```

# SETS

*A set is an efficient data structure for storing and processing nonduplicate elements. A map is like a dictionary that provides a quick lookup to retrieve a value using a key.*

The “**No-Fly**” **list** is a list, created and maintained by the U.S. government’s Terrorist Screening Center, of people who are not permitted to board a commercial aircraft for travel in or out of the United States. Suppose we need to write a program that checks whether a person is on the No-Fly list. You can use a list to store names in the No-Fly list. However, a more efficient data structure for this application is a *set*.

Suppose your program also needs to store detailed information about terrorists in the No-Fly list. The detailed information such as gender, height, weight, and nationality can be retrieved using the name as the key. A *map* is an efficient data structure for such a task.

*You can create a set using one of its three concrete classes:* **HashSet***,* **LinkedHashSet***, or* **TreeSet***.*

The **Set** interface extends the **Collection** interface, as shown in Figure 20.1. It does not introduce new methods or constants, but it stipulates that an instance of **Set** contains no duplicate elements. The concrete classes that implement **Set** must ensure that no duplicate elements can be added to the set.

The **AbstractSet** class extends **AbstractCollection** and partially implements **Set**. The **AbstractSet** class provides concrete implementations for the **equals** method and the **hashCode** method. The hash code of a set is the sum of the hash codes of all the elements in the set. Since the **size** method and **iterator** method are not implemented in the **AbstractSet** class, **AbstractSet** is an abstract class.

Three concrete classes of **Set** are **HashSet**, **LinkedHashSet**, and **TreeSet**.

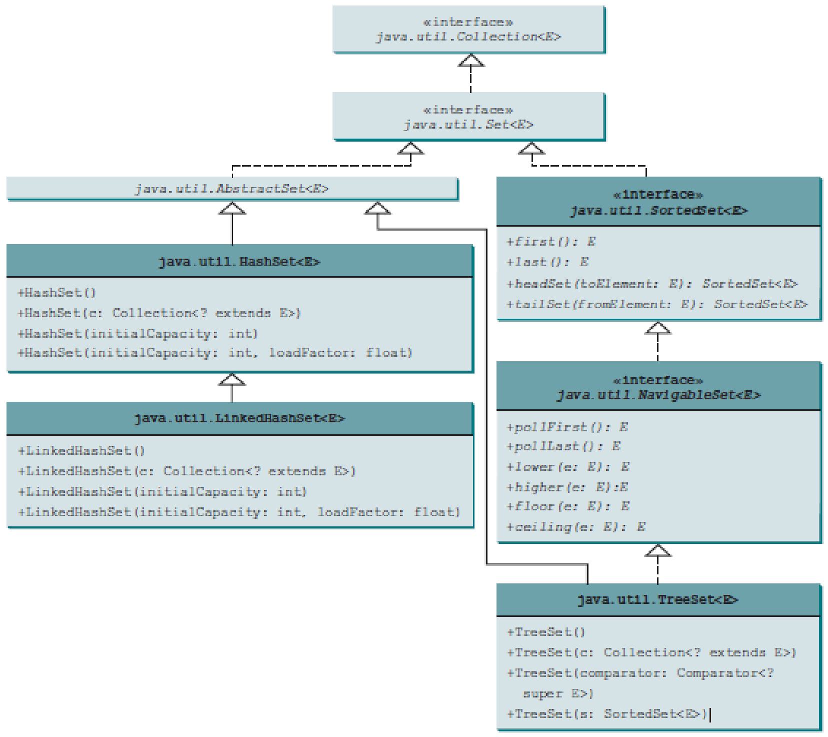
## HashSet

The **HashSet** class is a concrete class that implements **Set**. You can create an empty *hash set* using its no-arg constructor, or create a hash set from an existing collection. By default, the initial capacity is **16** and the load factor is **0.75**. If you know the size of your set, you can specify the initial capacity and load factor in the constructor. Otherwise, use the default setting. The load factor is a value between **0.0** and **1.0**.

*The load factor* measures how full the set is allowed to be before its capacity is increased. When the number of elements exceeds the product of the capacity and load factor, the capacity is automatically doubled. For example, if the capacity is **16** and load factor is **0.75**, the capacity will be doubled to **32** when the size reaches **12** (16 * 0.75 = 12). A higher load factor decreases the space costs but increases the search time. Generally, the default load factor **0.75** is a good tradeoff between time and space costs.

A **HashSet** can be used to store *duplicate-free* elements. For efficiency, objects added to a hash set need to implement the **hashCode** method in a manner that properly disperses the hash code. The **hashCode** method is defined in the **Object** class. The hash codes of two objects must be the same if the two objects are equal. Two unequal objects may have the same hash code, but you should implement the **hashCode** method to avoid too many such cases. Most of the classes in the Java API implement the **hashCode** method. For example, the **hashCode** in the **Integer** class returns its **int** value. The **hashCode** in the **Character** class returns the Unicode of the character. The **hashCode** in the **String** class returns s0 * 31(n-1) + s1 * 31(n-2) + g + sn-1, where s i is **s.charAt(i)**.

**Set** does not store duplicate elements. Two elements **e1** and **e2** are considered duplicate for a **HashSet** if **e1.equals(e2)** is true and **e1.hashCode() == e2.hashCode()**. Note that by contract, if two elements are equal, their **hashCode** must be same. So you need to override the **hashCode()** method whenever the **equals** method is overridden in the class.


«interface» java.util.Collection<E>
«interface» java.util.Set<E>
java.util.AbstractSet<E>
«interface» java.util.SortedSet<E>
+first(): E
+last(): E
+headSet(toElement: E): SortedSet<E>
+tailSet(fromElement: E): SortedSet<E>
java.util.HashSet<E>
+HashSet()
+HashSet(c: Collection<? extends E>)
+HashSet(initialCapacity: int)
+HashSet(initialCapacity: int, loadFactor: float)
java.util.LinkedHashSet<E>
+LinkedHashSet()
+LinkedHashSet(c: Collection<? extends E>)
+LinkedHashSet(initialCapacity: int)
+LinkedHashSet(initialCapacity: int, loadFactor: float)
«interface» java.util.NavigableSet<E>
+pollFirst(): E
+pollLast(): E
+lower(e: E): E
+higher(e: E):E
+floor(e: E): E
+ceiling(e: E): E
java.util.TreeSet<E>
+TreeSet()
+TreeSet(c: Collection<? extends E>)
+TreeSet(comparator: Comparator<? super E>)
+TreeSet(s: SortedSet<E>)

TestHashSet.java gives a program that creates a hash set to store strings and uses a foreach loop and a **forEach** method to traverse the elements in the set.

TestHashSet.java gives a program that creates a hash set to store strings and uses a foreach loop and a **forEach** method to traverse the elements in the set.

**TestHashSet.java**

```
import java.util.*;

public class TestHashSet {
    public static void main(String[] args) {
        // Create a hash set
        Set<String> set = new HashSet<>();
        // Add strings to the set
        set.add("London");
        set.add("Paris");
        set.add("New York");
        set.add("San Francisco");
        set.add("Beijing");
        set.add("New York");

        System.out.println(set);

        // Display the elements in the hash set
        for (String s: set) {
            System.out.print(s.toUpperCase() + " ");
        }
        // Process the elements using a forEach method
        System.out.println();
        set.forEach(e -> System.out.print(e.toLowerCase() + " "));
    }
}
```

```
OUTPUT
[San Francisco, New York, Paris, Beijing, London]
SAN FRANCISCO NEW YORK PARIS BEIJING LONDON
```

The strings are added to the set (lines 9–14). **New York** is added to the set more than once, but only one string is stored because a set does not allow duplicates.

As shown in the output, the strings are not stored in the order in which they are inserted intothe set. There is no particular order for the elements in a hash set. To impose an order on them, you need to use the **LinkedHashSet** class.

Recall that the **Collection** interface extends the **Iterable** interface, so the elements in a set are iterable. A foreach loop is used to traverse all the elements in the set (lines 19–21).

You can also use a **forEach** method to process each element in a set (line 25).

Since a set is an instance of **Collection**, all methods defined in **Collection** can be used for sets. TestMethodsInCollection.java gives an example that applies the methods in the **Collection** interface on sets.

**TestMethodsInCollection.java**

```
1 public class TestMethodsInCollection {
2          public static void main(String[] args) {
3               // Create set1
4               java.util.Set<String> set1 = new java.util.HashSet<>();
5
6               // Add strings to set1
7               set1.add("London");
8               set1.add("Paris");
9               set1.add("New York");
10              set1.add("San Francisco");
11              set1.add("Beijing");
12
13              System.out.println("set1 is " + set1);
14              System.out.println(set1.size() + " elements in set1");
15
16              // Delete a string from set1
17              set1.remove("London");
18              System.out.println("\nset1 is " + set1);
19              System.out.println(set1.size() + " elements in set1");
20
21              // Create set2
22              java.util.Set<String> set2 = new java.util.HashSet<>();
23
24              // Add strings to set2
25              set2.add("London");
26              set2.add("Shanghai");
27              set2.add("Paris");
28              System.out.println("\nset2 is " + set2);
29              System.out.println(set2.size() + " elements in set2");
30
31              System.out.println("\nIs Taipei in set2? "
32              + set2.contains("Taipei"));
33
34              set1.addAll(set2);
35              System.out.println("\nAfter adding set2 to set1, set1 is "
36              + set1);
37
38              set1.removeAll(set2);
39              System.out.println("After removing set2 from set1, set1 is "
40              + set1);
41
42              set1.retainAll(set2);
43              System.out.println("After retaining common elements in set2 "
44              + "and set2, set1 is " + set1);
45          }
46 }
OUTPUT:
set1 is [San Francisco, New York, Paris, Beijing, London]
5 elements in set1
set1 is [San Francisco, New York, Paris, Beijing]
4 elements in set1
set2 is [Shanghai, Paris, London]
3 elements in set2
Is Taipei in set2? false
After adding set2 to set1, set1 is
[San Francisco, New York, Shanghai, Paris, Beijing, London]
After removing set2 from set1, set1 is
[San Francisco, New York, Beijing]
After retaining common elements in set1 and set2, set1 is []
```

The program creates two sets (lines 4 and 22). The **size()** method returns the number of the elements in a set (line 14). Line 17

```
set1.remove("London");
```

removes **London** from **set1**.
The **contains** method (line 32) checks whether an element is in the set.
Line 34

```
set1.addAll(set2);
```

adds **set2** to **set1**. Therefore, **set1** becomes **[San Francisco, New York, Shanghai, Paris, Beijing, London]**.
Line 38

```
set1.removeAll(set2);
```

removes **set2** from **set1**. Thus, **set1** becomes **[San Francisco, New York, Beijing]**.
Line 42

```
set1.retainAll(set2);
```

retains the common elements in **set1** and **set2**. Since **set1** and **set2** have no common elements, **set1** becomes empty.

## LinkedHashSet

**LinkedHashSet** extends **HashSet** with a linked-list implementation that supports an ordering of the elements in the set. The elements in a **HashSet** are not ordered, but the elements in a **LinkedHashSet** can be retrieved in the order in which they were inserted into the set. A **LinkedHashSet** can be created by using one of its four constructors.

These constructors are similar to the constructors for **HashSet**. TestLinkedHashSet.java gives a test program for **LinkedHashSet**. The program simply replaces **HashSet** by **LinkedHashSet** in TestHashSet.java.

**TestLinkedHashSet.java**

```
import java.util.*;

public class TestLinkedHashSet {
        public static void main(String[] args) {
              // Create a hash set
              Set<String> set = new LinkedHashSet<>();

              // Add strings to the set
9               set.add("London");
10              set.add("Paris");
11              set.add("New York");
12              set.add("San Francisco");
13              set.add("Beijing");
14              set.add("New York");
15
16              System.out.println(set);
17
18              // Display the elements in the hash set
19              for (String element: set)
20                   System.out.print(element.toLowerCase() + " ");
21          }
22 }
OUTPUT
[London, Paris, New York, San Francisco, Beijing]
london paris new york san francisco beijing
```

A **LinkedHashSet** is created in line 6. As shown in the output, the strings are stored in the order in which they are inserted. Since **LinkedHashSet** is a set, it does not store duplicate elements.

The **LinkedHashSet** maintains the order in which the elements are inserted. To impose a different order (e.g., increasing or decreasing order), you can use the **TreeSet** class, which is introduced in the next section.

**If you don't need to maintain the order in which the elements are inserted, use **HashSet**, which is more efficient than **LinkedHashSet**.**

## TreeSet

**SortedSet** is a subinterface of **Set**, which guarantees that the elements in the set are sorted. In addition, it provides the methods **first()** and **last()** for returning the first and last elements in the set, and **headSet(toElement)** and **tailSet(-fromElement)** for returning a portion of the set whose elements are less than **toElement** and greater than or equal to **fromElement**, respectively.

**NavigableSet** extends **SortedSet** to provide navigation methods **lower(e)**, **floor(e)**, **ceiling(e)**, and **higher(e)** that return elements, respectively, less than, less than or equal, greater than or equal, and greater than a given element and return **null** if there is no such element. The **pollFirst()** and **pollLast()** methods remove and return the first and last element in the tree set, respectively.

**TreeSet** implements the **SortedSet** interface. To create a **TreeSet**, use a constructor. You can add objects into a *tree set* as long as they can be compared with each other.

The elements can be compared in two ways: using the **Comparable** interface or the **Comparator** interface.

TestTreeSet.java gives an example of ordering elements using the **Comparable** interface. The preceding example in TestLinkedHashSet.java displays all the strings in their insertion order. This example rewrites the preceding example to display the strings in alphabetical order using the **TreeSet** class.

**TestTreeSet.java**

```
1 import java.util.*;
2
3 public class TestTreeSet {
4          public static void main(String[] args) {
5               // Create a hash set
6               Set<String> set = new HashSet<>();
7
8               // Add strings to the set
9               set.add("London");
10              set.add("Paris");
11              set.add("New York");
12              set.add("San Francisco");
13              set.add("Beijing");
14              set.add("New York");
15
16              TreeSet<String> treeSet = new TreeSet<>(set);
17              System.out.println("Sorted tree set: " + treeSet);
18
19              // Use the methods in SortedSet interface
20              System.out.println("first(): " + treeSet.first());
21              System.out.println("last(): " + treeSet.last());
22              System.out.println("headSet(\"New York\"): " +
23              treeSet.headSet("New York"));
24              System.out.println("tailSet(\"New York\"): " +
25              treeSet.tailSet("New York"));
26
27              // Use the methods in NavigableSet interface
28              System.out.println("lower(\"P\"): " + treeSet.lower("P"));
29              System.out.println("higher(\"P\"): " + treeSet.higher("P"));
30              System.out.println("floor(\"P\"): " + treeSet.floor("P"));
31              System.out.println("ceiling(\"P\"): " + treeSet.ceiling("P"));
32              System.out.println("pollFirst(): " + treeSet.pollFirst());
33              System.out.println("pollLast(): " + treeSet.pollLast());
34              System.out.println("New tree set: " + treeSet);
35          }
36 }
OUTPUT:
Sorted tree set: [Beijing, London, New York, Paris, San Francisco]
first(): Beijing
last(): San Francisco
headSet("New York"): [Beijing, London]
tailSet("New York"): [New York, Paris, San Francisco]
lower("P"): New York
higher("P"): Paris
floor("P"): New York
ceiling("P"): Paris
pollFirst(): Beijing
pollLast(): San Francisco
New tree set: [London, New York, Paris]
```

The example creates a hash set filled with strings, then creates a tree set for the same strings. The strings are sorted in the tree set using the **compareTo** method in the **Comparable** interface. Two elements **e1** and **e2** are considered duplicate for a **TreeSet** if **e1.compareTo(e2)** is **0** for **Comparable** and **e1.compare(e2)** is **0** for **Comparator**.

The elements in the set are sorted once you create a **TreeSet** object from a **HashSet** object using **new TreeSet<>(set)** (line 16). You may rewrite the program to create an instance of **TreeSet** using its no-arg constructor and add the strings into the **TreeSet** object.

**treeSet.first()** returns the first element in **treeSet** (line 20) and **treeSet.last()** returns the last element in **treeSet** (line 21). **treeSet.headSet("New York")** returns the elements in **treeSet** before New York (lines 22–23). **treeSet.tailSet("New York")** returns the elements in **treeSet** after New York, including New York (lines 24–25).

**treeSet.lower("P")** returns the largest element less than **P** in **treeSet** (line 28). **treeSet.higher("P")** returns the smallest element greater than **P** in **treeSet** (line 29). **treeSet.floor("P")** returns the largest element less than or equal to **P** in **treeSet** (line 30). **treeSet.ceiling("P")** returns the smallest element greater than or equal to **P** in **treeSet** (line 31). **treeSet.pollFirst()** removes the first element in **treeSet** and returns the removed element (line 32). **treeSet.pollLast()** removes the last element in **treeSet** and returns the removed element (line 33).

_All the concrete classes in Java Collections Framework have at least two constructors. One is the no-arg constructor that constructs an empty collection. The other constructs instances from a collection. Thus the **TreeSet** class has the constructor **TreeSet(Collection c)** for constructing a **TreeSet** from a collection **c**. In this example, **new TreeSet<>(set)** creates an instance of **TreeSet** from the collection **set**._

_*If you don’t need to maintain a sorted set when updating a set, you should use a hash set because it takes less time to insert and remove elements in a hash set. When you need a sorted set, you can create a tree set from the hash set.*_

If you create a **TreeSet** using its no-arg constructor, the **compareTo** method is used to compare the elements in the set, assuming the class of the elements implements the **Comparable** interface. To use a comparator, you have to use the constructor **TreeSet(Comparator comparator)** to create a sorted set that uses the **compare** method in the comparator to order the elements in the set.

TestTreeSetWithComparator.java gives a program that demonstrates how to sort elements in a tree set using the **Comparator** interface.

```
TestTreeSetWithComparator.java
1 import java.util.*;
2
3 public class TestTreeSetWithComparator {
4         public static void main(String[] args) {
5              // Create a tree set for geometric objects using a comparator
6              Set<GeometricObject> set =
7              new TreeSet<>(new GeometricObjectComparator());
8              set.add(new Rectangle(4, 5));
9              set.add(new Circle(40));
10             set.add(new Circle(40));
11             set.add(new Rectangle(4, 1));
12
13             // Display geometric objects in the tree set
14             System.out.println("A sorted set of geometric objects");
15             for (GeometricObject element: set)
16             System.out.println("area = " + element.getArea());
17        }
18 }
OUTPUT
A sorted set of geometric objects
area = 4.0
area = 20.0
area = 5021.548245743669
```

The **GeometricObjectComparator** class is defined in Listing 20.4. The program creates a tree set of geometric objects using the **GeometricObjectComparator** for comparing the elements in the set (lines 6 and 7).

The **Circle** and **Rectangle** classes were defined in Section 13.2, Abstract Classes. They are all subclasses of **GeometricObject**. They are added to the set (lines 8–11).

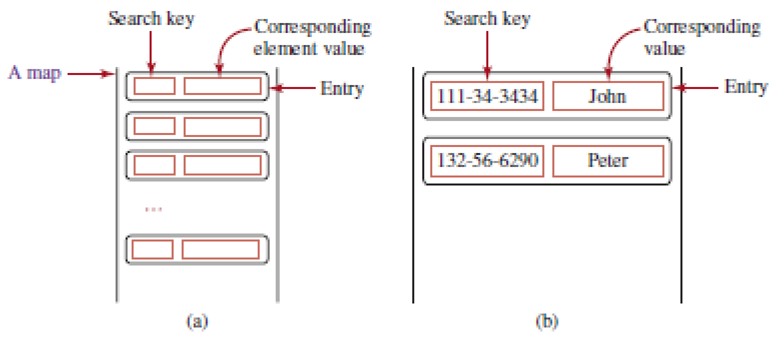
Two circles of the same radius are added to the tree set (lines 9 and 10), but only one is stored because the two circles are equal (determined by the comparator in this case) and the set does not allow duplicates.

# MAPS

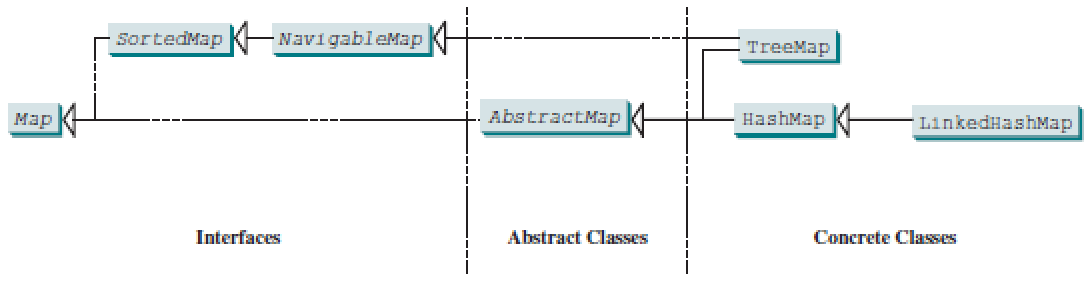
*You can create a map using one of its three concrete classes:* **HashMap***,* **LinkedHashMap***, or* **TreeMap***.*

A *map* is a container object that stores a collection of key/value pairs. It enables fast retrieval, deletion, and updating of the pair through the key. A map stores the values along with the keys.

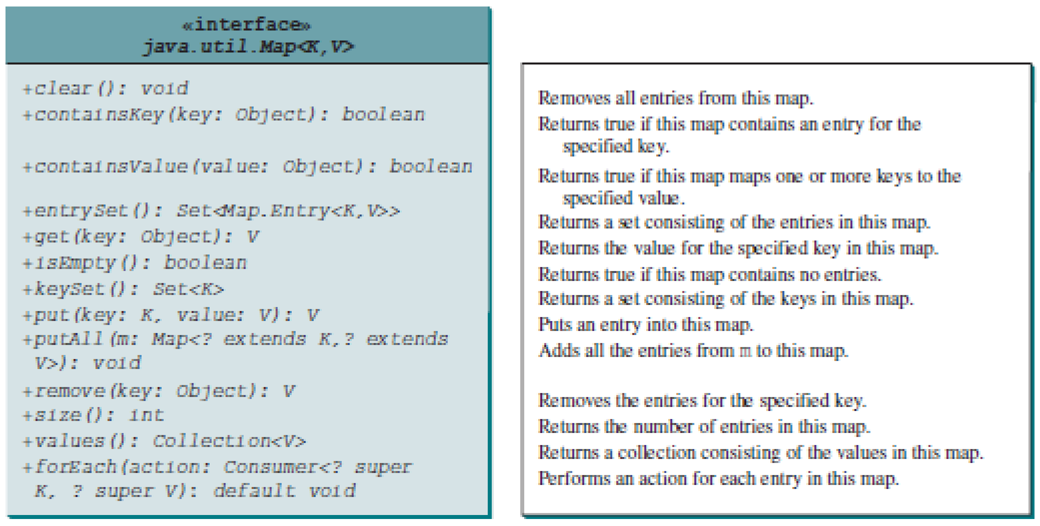
The keys are like indexes. In **List**, the indexes are integers. In **Map**, the keys can be any objects. A map cannot contain duplicate keys. Each key maps to one value. A key and its corresponding value form an entry stored in a map, as shown in the figure below (a). In the figure below (a) shows a map in which each entry consists of a Social Security number as the key and a name as the value.



There are three types of maps: **HashMap**, **LinkedHashMap**, and **TreeMap**. The common features of these maps are defined in the **Map** interface. Their relationship is shown in the figure below:



The **Map** interface provides the methods for querying, updating, and obtaining a collection of values and a set of keys, as shown in the figure below:

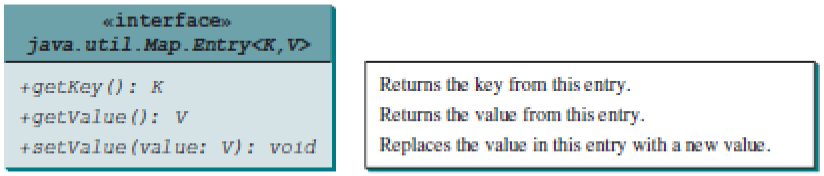


| «interface» java.util.Map<K,V> | |
|---|---|
| +clear(): void | Removes all entries from this map. |
| +containsKey(key: Object): boolean | Returns true if this map contains an entry for the specified key. |
| +containsValue(value: Object): boolean | Returns true if this map maps one or more keys to the specified value. |
| +entrySet(): Set<Map.Entry<K,V>> | Returns a set consisting of the entries in this map. |
| +get(key: Object): V | Returns the value for the specified key in this map. |
| +isEmpty(): boolean | Returns true if this map contains no entries. |
| +keySet(): Set<K> | Returns a set consisting of the keys in this map. |
| +put(key: K, value: V): V | Puts an entry into this map. |
| +putAll(m: Map<? extends K,? extends V>): void | Adds all the entries from m to this map. |
| +remove(key: Object): V | Removes the entries for the specified key. |
| +size(): int | Returns the number of entries in this map. |
| +values(): Collection<V> | Returns a collection consisting of the values in this map. |
| +forEach(action: Consumer<? super K, ? super V>): default void | Performs an action for each entry in this map. |

The *update methods* include **clear**, **put**, **putAll**, and **remove**. The **clear()** method removes all entries from the map. The **put(K key, V value)** method adds an entry for the specified key and value in the map. If the map formerly contained an entry for this key, the old value is replaced by the new value, and the old value associated with the key is returned. The **putAll(Map m)** method adds all entries in **m** to this map. The **remove(Object key)** method removes the entry for the specified key from the map.

The *query methods* include **containsKey**, **containsValue**, **isEmpty**, and **size**. The **containsKey(Object key)** method checks whether the map contains an entry for the specified key. The **containsValue(Object value)** method checks whether the map contains an entry for this value. The **isEmpty()** method checks whether the map contains any entries. The **size()** method returns the number of entries in the map.

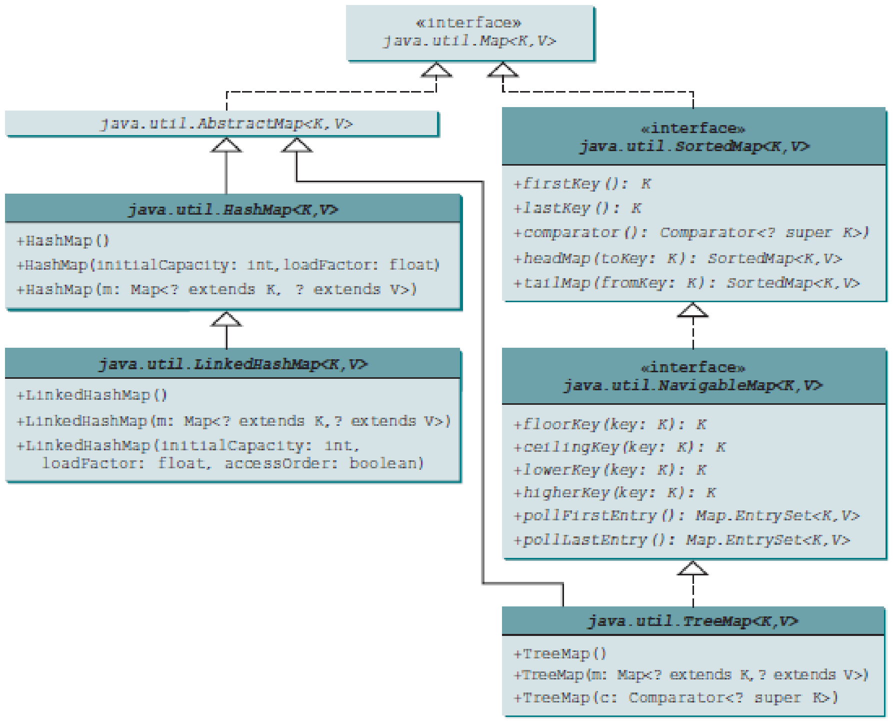
You can obtain a set of the keys in the map using the **keySet()** method, and a collection of the values in the map using the **values()** method. The **entrySet()** method returns a set of entries. The entries are instances of the **Map.Entry<K, V>** interface, where **Entry** is an inner interface for the **Map** interface, as shown in the figure below. Each entry in the set is a key/value pair in the underlying map.


«interface»
java.util.Map.Entry<K,V>
+getKey(): K
+getValue(): V
+setValue(value: V): void
Returns the key from this entry.
Returns the value from this entry.
Replaces the value in this entry with a new value.

Java 8 added a default **forEach** method in the **Map** interface for performing an action on each entry in the map. This method can be used like an iterator for traversing the entries in the map.

The **AbstractMap** class is a convenience abstract class that implements all the methods in the **Map** interface except the **entrySet()** method.

The **HashMap**, **LinkedHashMap**, and **TreeMap** classes are three *concrete implementations* of the **Map** interface, as shown in the figure below:


«interface»
java.util.Map<K,V>
java.util.AbstractMap<K,V>
«interface»
java.util.SortedMap<K,V>
+firstKey(): K
+lastKey(): K
+comparator(): Comparator<? super K>)
+headMap(toKey: K): SortedMap<K,V>
+tailMap(fromKey: K): SortedMap<K,V>
java.util.HashMap<K,V>
+HashMap()
+HashMap(initialCapacity: int, loadFactor: float)
+HashMap(m: Map<? extends K, ? extends V>)
java.util.LinkedHashMap<K,V>
+LinkedHashMap()
+LinkedHashMap(m: Map<? extends K,? extends V>)
+LinkedHashMap(initialCapacity: int, loadFactor: float, accessOrder: boolean)
«interface»
java.util.NavigableMap<K,V>
+floorKey(key: K): K
+ceilingKey(key: K): K
+lowerKey(key: K): K
+higherKey(key: K): K
+pollFirstEntry(): Map.EntrySet<K,V>
+pollLastEntry(): Map.EntrySet<K,V>
java.util.TreeMap<K,V>
+TreeMap()
+TreeMap(m: Map<? extends K,? extends V>)
+TreeMap(c: Comparator<? super K>)

The **HashMap** class is efficient for locating a value, inserting an entry, and deleting an entry.

**LinkedHashMap** extends **HashMap** with a linked-list implementation that supports an ordering of the entries in the map. The entries in a **HashMap** are not ordered, but the entries in a **LinkedHashMap** can be retrieved either in the order in which they were inserted into the map (known as the *insertion order*) or in the order in which they were last accessed, from least recently to most recently accessed (*access order*). The no-arg constructor constructs a **LinkedHashMap** with the insertion order. To construct a **LinkedHashMap** with the access order, use **LinkedHashMap(initialCapacity, loadFactor, true)**.

The **TreeMap** class is efficient for traversing the keys in a sorted order. The keys can be sorted using the **Comparable** interface or the **Comparator** interface. If you create a **TreeMap** using its no-arg constructor, the **compareTo** method in the **Comparable** interface is used to compare the keys in the map, assuming the class for the keys implements the **Comparable** interface. To use a comparator, you have to use the **TreeMap(Comparator comparator)** constructor to create a sorted map that uses the **compare** method in the comparator to order the entries in the map based on the keys.

**SortedMap** is a subinterface of **Map**, which guarantees the entries in the map are sorted. In addition, it provides the methods **firstKey()** and **lastKey()** for returning the first and the last keys in the map, and **headMap(toKey)** and **tailMap(fromKey)** for returning a portion of the map whose keys are less than **toKey** and greater than or equal to **fromKey**, respectively.

**NavigableMap** extends **SortedMap** to provide the navigation methods **lowerKey(key)**, **floorKey(key)**, **ceilingKey(key)**, and **higherKey(key)** that return keys, respectively, less than, less than or equal, greater than or equal, and greater than a given key and return **null** if there is no such key. The **pollFirstEntry()** and **pollLastEntry()** methods remove and return the first and the last entry in the tree map, respectively.

**Prior to Java 2, **java.util.Hashtable** was used for mapping keys with values. **Hashtable** was redesigned to fit into the Java Collections Framework with all its methods retained for compatibility. **Hashtable** implements the **Map** interface and is used in the same way as **HashMap**, except that the update methods in **Hashtable** are synchronized.**

TestMap.java gives an example that creates a *hash map*, a *linked hash map*, and a *tree map* for mapping students to ages. The program first creates a hash map with the student's name as its key and the age as its value. The program then creates a tree map from the hash map and displays the entries in ascending order of the keys. Finally, the program creates a linked hash map, adds the same entries to the map, and displays the entries.

```
TestMap.java
import java.util.*;

public class TestMap {
        public static void main(String[] args) {
             // Create a HashMap
             Map<String, Integer> hashMap = new HashMap<>();
             hashMap.put("Smith", 30);
             hashMap.put("Anderson", 31);
             hashMap.put("Lewis", 29);
             hashMap.put("Cook", 29);

             System.out.println("Display entries in HashMap");
             System.out.println(hashMap + "\n");

             // Create a TreeMap from the preceding HashMap
             Map<String, Integer> treeMap = new TreeMap<>(hashMap);
             System.out.println("Display entries in ascending order of key");
             System.out.println(treeMap);

             // Create a LinkedHashMap
             Map<String, Integer> linkedHashMap =
                  new LinkedHashMap<>(16, 0.75f, true);
             linkedHashMap.put("Smith", 30);
             linkedHashMap.put("Anderson", 31);
             linkedHashMap.put("Lewis", 29);
             linkedHashMap.put("Cook", 29);

             // Display the age for Lewis
29              System.out.println("\nThe age for " + "Lewis is " +
30              linkedHashMap.get("Lewis"));
31
32              System.out.println("Display entries in LinkedHashMap");
33              System.out.println(linkedHashMap);
34
35               // Display each entry with name and age
36              System.out.print("\nNames and ages are ");
37              treeMap.forEach(
38                   (name, age) -> System.out.print(name + ": " + age + " "));
39          }
40 }
```

**OUTPUT:**

```
Display entries in HashMap
{Cook=29, Smith=30, Lewis=29, Anderson=31}
Display entries in ascending order of key
{Anderson=31, Cook=29, Lewis=29, Smith=30}
The age for Lewis is 29
Display entries in LinkedHashMap
{Smith=30, Anderson=31, Cook=29, Lewis=29}
Names and ages are Anderson: 31 Cook: 29 Lewis: 29 Smith: 30
```

As shown in the output, the entries in the **HashMap** are in random order. The entries in the **TreeMap** are in increasing order of the keys. The entries in the **LinkedHashMap** are in the order of their access, from least recently accessed to most recently.

All the concrete classes that implement the **Map** interface have at least two constructors. One is the no-arg constructor that constructs an empty map, and the other constructs a map from an instance of **Map**. Thus, **new TreeMap<>(hashMap)** (line 16) constructs a tree map from a hash map.

You can create an insertion- or access-ordered linked hash map. An access-ordered linked hash map is created in lines 21–22. The most recently accessed entry is placed at the end of the map. The entry with the key **Lewis** is last accessed in line 30, so it is displayed last in line 33.

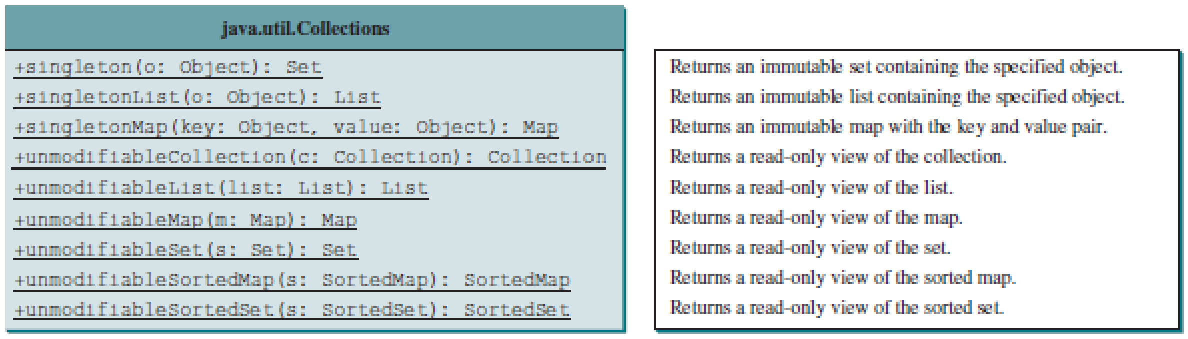
It is convenient to process all the entries in the map using the **forEach** method. The program uses a **forEach** method to display a name and its age (lines 37–38).

***If you don't need to maintain an order in a map when updating it, use a HashMap. When you need to maintain the insertion order or access order in the map, use a LinkedHashMap. When you need the map to be sorted on keys, use a TreeMap.***

## Singleton and Unmodifiable Collections and Maps

*You can create singleton sets, lists, and maps and unmodifiable sets, lists, and maps using the static methods in the* **Collections** *class.*

The **Collections** class contains the static methods for lists and collections. It also contains the methods for creating immutable singleton sets, lists, and maps and for creating read-only sets, lists, and maps, as shown in the figure below:



| java.util.Collections | |
|---|---|
| +singleton(o: Object): Set | Returns an immutable set containing the specified object. |
| +singletonList(o: Object): List | Returns an immutable list containing the specified object. |
| +singletonMap(key: Object, value: Object): Map | Returns an immutable map with the key and value pair. |
| +unmodifiableCollection(c: Collection): Collection | Returns a read-only view of the collection. |
| +unmodifiableList(list: List): List | Returns a read-only view of the list. |
| +unmodifiableMap(m: Map): Map | Returns a read-only view of the map. |
| +unmodifiableSet(s: Set): Set | Returns a read-only view of the set. |
| +unmodifiableSortedMap(s: SortedMap): SortedMap | Returns a read-only view of the sorted map. |
| +unmodifiableSortedSet(s: SortedSet): SortedSet | Returns a read-only view of the sorted set. |

The **Collections** class defines three constants—**EMPTY_SET**, **EMPTY_LIST**, and **EMPTY_MAP**—for an empty set, an empty list, and an empty map. These collections are immutable. The class also provides the **singleton(Object o)** method for creating an immutable set containing only a single item, the **singletonList(Object o)** method for creating an immutable list containing only a single item, and the **singletonMap(Object key, Object value)** method for creating an immutable map containing only a single entry.

The **Collections** class also provides six static methods for returning *read-only views for collections*: **unmodifiableCollection(Collection c)**, **unmodifiableList(List list)**, **unmodifiableMap(Map m)**, **unmodifiableSet(Set set)**, **unmodifiable- SortedMap(SortedMap m)**, and **unmodifiableSortedSet(SortedSet s)**. This type of view is like a reference to the actual collection. However, you cannot modify the collection through a read-only view. Attempting to modify a collection through a read-only view will cause an **UnsupportedOperationException**.

In JDK 9, you can use the static **Set.of(e1, e2, ...)** method to create an immutable set and **Map.of(key1, value1, key2, value2, ...)** method to create an immutable map.

Source:

Introduction to Java$^{TM}$ Programming and Data Structures, *Comprehensive Version* **Y. Daniel Liang** TWELFTH EDITION GLOBAL EDITION, 2022, Pearson, Chapter 20 & Chapter 21.