Collections Framework

The 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 that provide state-of-the-art technology for managing groups of objects. It merits close attention by all programmers. Because java.util contains a wide array of functionality, it is quite large.

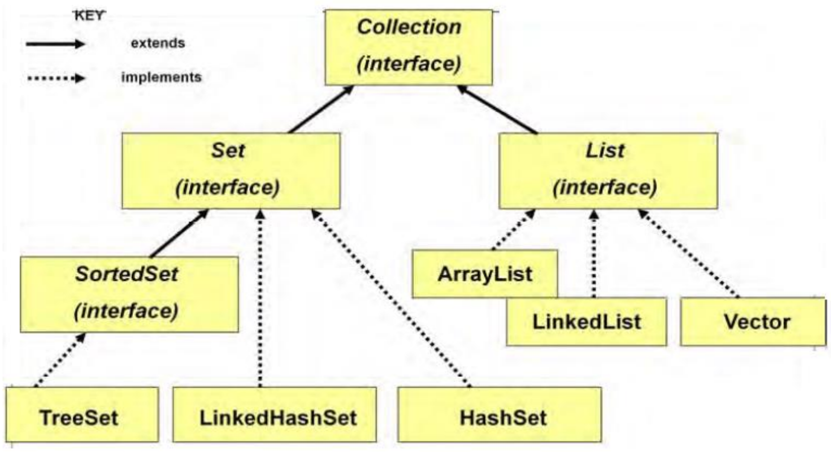
Collections Overview

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

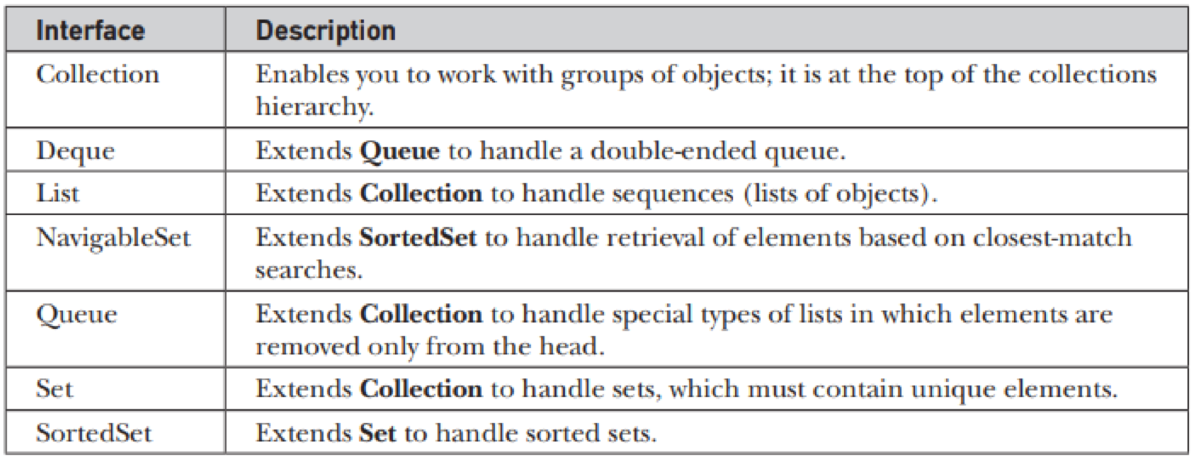
Benefits of the Java Collections Framework

- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level work required to make it work.
- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms.
- **Allows interoperability among unrelated APIs:** APIs will interoperate seamlessly w.r.to. collections, even though they were written independently.
- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output. With the advent of standard collection interfaces, the effort to learn and to use new APIs is reduced.
- **Reduces effort to design new APIs:** An API that relies on collections can easily be designed using standard collection interfaces.
- **Promotes software reuse:** new data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

<u>The Collection Interfaces</u>



The Collections Framework defines several core interfaces. This section provides an overview of each interface. Beginning with the collection interfaces is necessary because they determine the fundamental nature of the collection classes. Put differently, the concrete classes simply provide different implementations of the standard interfaces. The interfaces that underpin collections are summarized in the following table:



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.
List	Extends **Collection** to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches.
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.

In addition to the collection interfaces, collections also use the **Comparator, RandomAccess, Iterator, and ListIterator interfaces.** Beginning with JDK 8, **Spliterator** can also be used. Briefly, Comparator defines how two objects are compared; Iterator, ListIterator, and Spliterator enumerate the objects within a collection. By implementing RandomAccess, a list indicates that it supports efficient, random access to its elements. To provide the greatest flexibility in their use, the collection interfaces allow some methods to be optional. The optional methods enable you to modify 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. The following sections examine the collection interfaces.

The Collection Interface

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. (Recall that only classes that implement Iterable can be cycled through by the for.)

Collection declares the core methods that all collections will have. These methods are summarized in Table 18-1. 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.

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 collection changed (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.
default Stream<E> parallelStream( )	Returns a stream that uses the invoking collection as its source for elements. If possible, the stream supports parallel operations. (Added by JDK 8.)
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**.
default boolean removeIf(Predicate<? super E> *predicate*)	Removes from the invoking collection those elements that satisfy the condition specified by *predicate*. (Added by JDK 8.)

Table 18-1 The Methods Declared by **Collection**

Method	**Description**
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.
default Spliterator<E> spliterator( )	Returns a spliterator to the invoking collections. (Added by JDK 8.)
default Stream<E> stream( )	Returns a stream that uses the invoking collection as its source for elements. The stream is sequential. (Added by JDK 8.)
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 18-1 The Methods Declared by **Collection** *(continued)*

Note: Any 6

Objects are added to a collection by calling add( ). Notice that add( ) 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.

- You can add the entire contents of one collection to another by calling addAll( ).
- You can remove an object by using remove( ).
- To remove a group of objects, call removeAll( ).
- You can remove all elements except those of a specified group by calling retainAll( ).
- Beginning with JDK 8, to remove an element only if it satisfies some condition, you can use removeIf( ).
- To empty a collection, call clear( ).
- You can determine whether a collection contains a specific object by calling contains( ).
- To determine whether one collection contains all the members of another, call containsAll( ).
- You can determine when a collection is empty 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. Normally, the second form is more convenient because it returns the desired array type. These methods are more important than it might at first seem. Often, processing the contents of a collection by using array-like syntax is advantageous. By providing a pathway between collections and arrays, you can have the best of both worlds.
- Two collections can be compared for equality by calling equals( ). The precise meaning of "equality" may differ from collection to collection. For example, you can implement equals( ) so that it compares the values of elements stored in the collection. Alternatively, equals( ) can compare references to those elements.

- Another important method is iterator( ), which returns an iterator to a collection.

- The new spliterator( ) method returns a spliterator to the collection.
- Iterators are frequently used when working with collections.

- Finally, the stream( ) and parallelStream( ) methods return a Stream that uses the collection as a source of elements

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 Table 18-2. Note again that 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. Also, several 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. Also, 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.

- You can modify each element in the collection by using replaceAll( ).
- 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( ).
- You can obtain a sublist of a list by calling subList( ), specifying the beginning and ending indexes of the sublist. As you can imagine, subList( ) makes list processing quite convenient.
- One way to sort a list is with the sort( ) method defined by List.

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.
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.
default void replaceAll(UnaryOperator<E> *opToApply*)	Updates each element in the list with the value obtained from the *opToApply* function. (Added by JDK 8.)
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list. Returns the old value.
default void sort(Comparator<? super E> *comp*)	Sorts the list using the comparator specified by *comp*. (Added by JDK 8.)
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 18-2 The Methods Declared by **List**

Note: Any 5

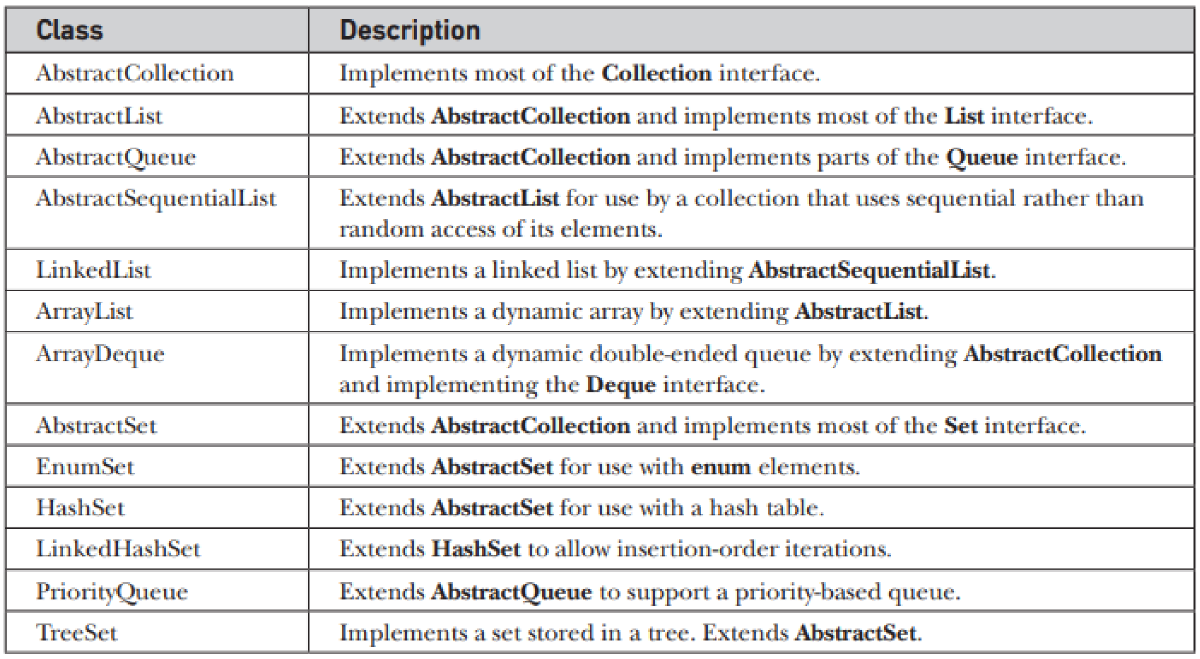
The Set Interface

The Set interface defines a set. It extends Collection and specifies 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 specify 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.

The Collection Classes

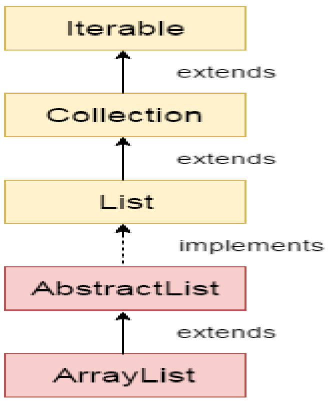
Now that you are familiar with the collection interfaces, you are ready to examine the standard classes that implement them. Some of the classes provide full implementations that can be used as-is. Others are abstract, providing skeletal implementations that are used as starting points for creating concrete collections. As a general rule, the collection classes are not synchronized, but as you will see later in this chapter, it is possible to obtain synchronized versions



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.
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**.

The following sections examine the concrete collection classes and illustrate their use.
NOTE In addition to the collection classes, several legacy classes, such as Vector, Stack, and Hashtable, have been reengineered to support 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 you must know in advance how many elements an array will hold. But, sometimes, you may not know until run time precisely how large an array you need. To handle this situation, the Collections Framework defines ArrayList.

In essence, 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)

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 array list.

The following program shows a simple use of ArrayList. An array list is created for objects of type String, and then several strings are added to it. (Recall 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.

```
// Demonstrate ArrayList.
import java.util.*;
class ArrayListDemo {
 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);
 }
}
```

The output from this program is shown here:

```
 Initial size of al: 0
 Size of al after additions: 7
 Contents of al: [C, A2, A, E, B, D, F]
 Size of al after deletions: 5
 Contents of al: [C, A2, E, B, D]
```

Obtaining an Array from an ArrayList

When working with ArrayList, you will sometimes want to obtain an actual array that contains the contents of the list. You can do this by calling toArray( ), which is defined by Collection. Several reasons exist why you might want to convert a collection into an array, such as:

- To obtain faster processing times for certain operations
- To pass an array to a method that is not overloaded to accept a collection
- To integrate collection-based code with legacy code that does not understand collections

```
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. Normally, the second form is more convenient because it returns the proper type of array. The following program demonstrates its use:

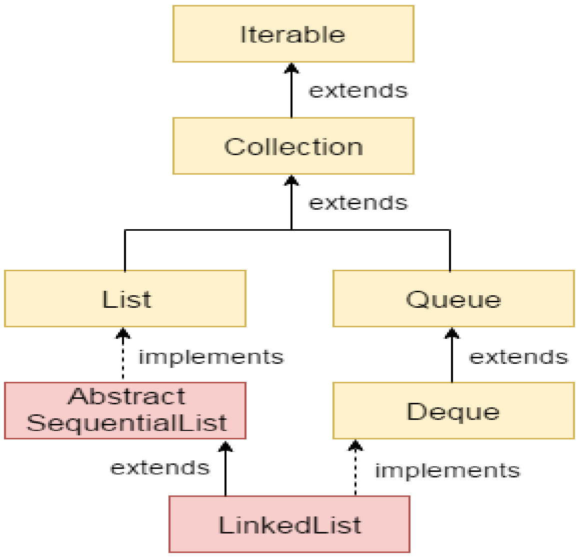
```
// Convert an ArrayList into an array.
import java.util.*;
class ArrayListToArray {
 public static void main(String args[]) {
 // Create an array list.
 ArrayList<Integer> al = new ArrayList<Integer>();
 // Add elements to the array list.
 al.add(1);
 al.add(2);
 al.add(3);
 al.add(4);
 System.out.println("Contents 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 output from the program is shown here:

```
 Contents of al: [1, 2, 3, 4]
 Sum is: 10
```

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.

<u>The LinkedList Class</u>



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 shown here:

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. For example, to add elements to the start of a list, you can use **addFirst( ) or offerFirst( )**. To add elements to the end of the list, use **addLast( ) or offerLast( ).** To obtain the first element, you can use **getFirst( ) or peekFirst( ).** 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( ).**

The following program illustrates LinkedList:

```
// Demonstrate LinkedList.
import java.util.*;
class LinkedListDemo {
 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);
 // 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 = 11.get(2);
 ll.set(2, val + " Changed");
 System.out.println("ll after change: " + ll);
 }
}
```

The output from this program is shown here:

```
 Original contents of ll: [A, A2, F, B, D, E, C, Z]
 Contents of ll after deletion: [A, A2, D, E, C, Z]
 ll after deleting first and last: [A2, D, E, C]
 ll after change: [A2, D, E Changed, C]
```

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. 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 hash code is performed automatically—you never see the hash code itself. 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.) 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 super classes 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.
If you need sorted storage, then another collection, such as TreeSet, is a better choice. Here is an example that demonstrates HashSet:

```
// Demonstrate HashSet.
import java.util.*;
class HashSetDemo {
 public static void main(String args[]) {
 // Create a hash set.
 HashSet<String> hs = new HashSet<String>();
 // Add elements to the hash set.
 hs.add("Beta");
 hs.add("Alpha");
 hs.add("Eta");
 hs.add("Gamma");
 hs.add("Epsilon");
 hs.add("Omega");
 System.out.println(hs);
 }
}
```

The following is the output from this program:

```
[Gamma, Eta, Alpha, Epsilon, Omega, Beta]
```

As explained, the elements are not stored in sorted order, and the precise output may vary.

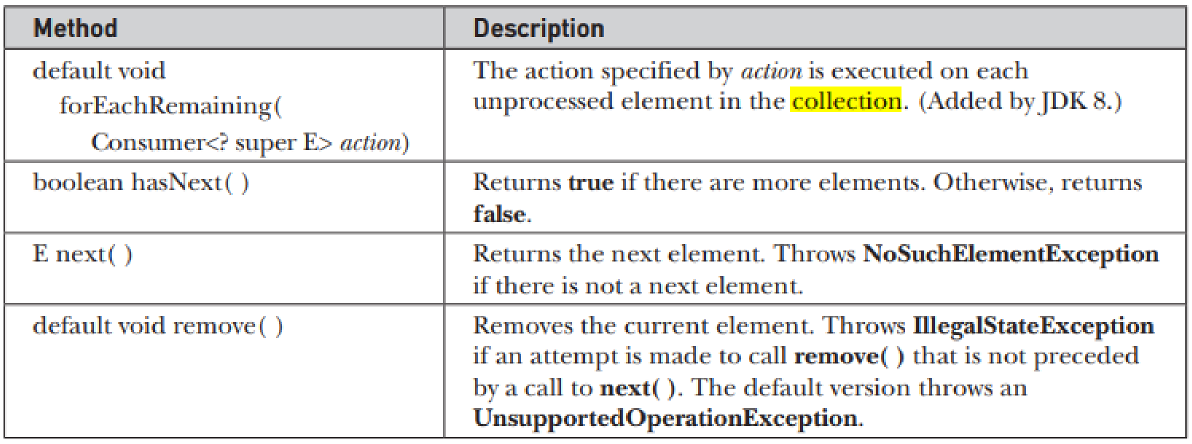
Accessing a Collection via an Iterator

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element. One way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface. Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements. Iterator and ListIterator are generic interfaces which are declared as shown here:

interface Iterator<E>

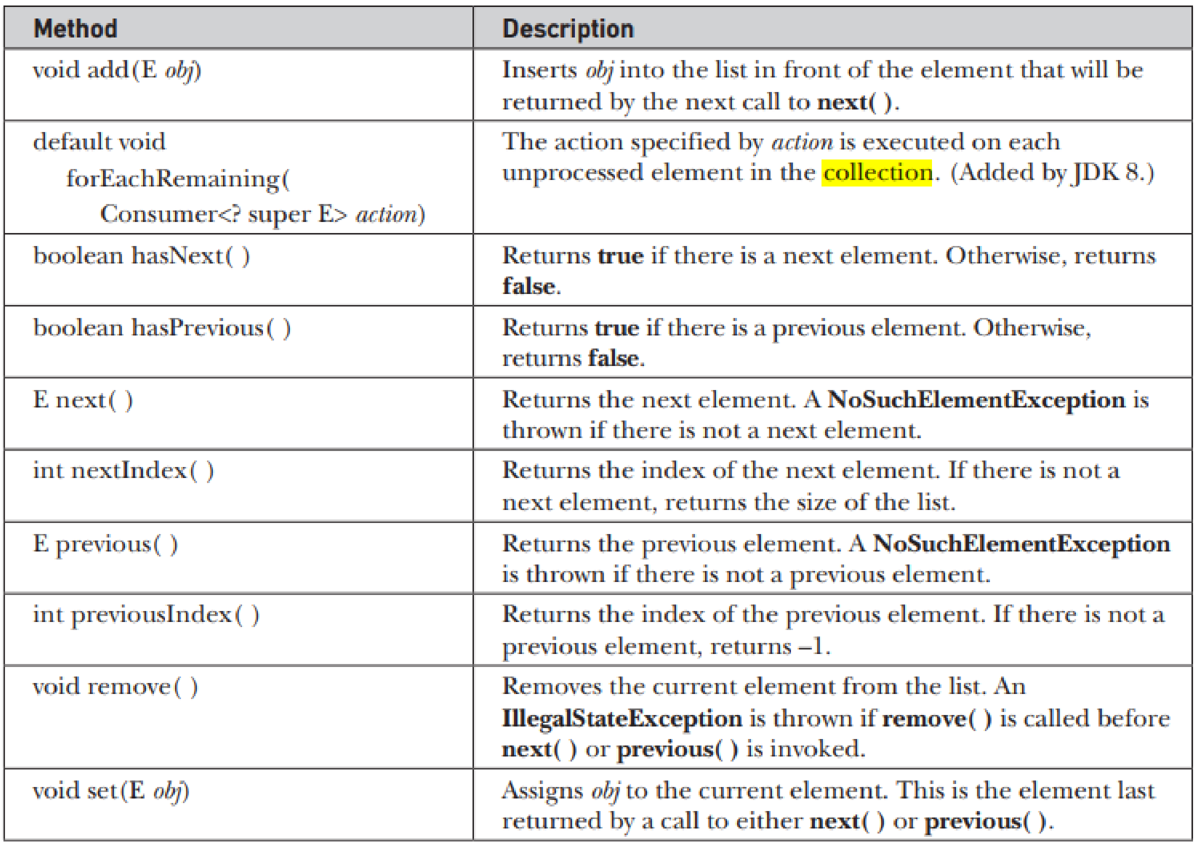
interface ListIterator<E>

Here, E specifies the type of objects being iterated. The Iterator interface declares the methods shown in Table 18-8. The methods declared by ListIterator (along with those inherited from Iterator) are shown in Table 18-9. In both cases, operations that modify the underlying collection are optional. For example, remove( ) will throw **UnsupportedOperationException** when used with a read-only collection. Various other exceptions are possible.



Method	Description
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection. (Added by JDK 8.)
boolean hasNext( )	Returns **true** if there are more elements. Otherwise, returns **false**.
E next( )	Returns the next element. Throws **NoSuchElementException** if there is not a next element.
default void remove( )	Removes the current element. Throws **IllegalStateException** if an attempt is made to call **remove( )** that is not preceded by a call to **next( )**. The default version throws an **UnsupportedOperationException**.

Table 18-8 The Methods Declared by **Iterator**



Method	Description
void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to **next( )**.
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection. (Added by JDK 8.)
boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**.
boolean hasPrevious( )	Returns **true** if there is a previous element. Otherwise, returns **false**.
E next( )	Returns the next element. A **NoSuchElementException** is thrown if there is not a next element.
int nextIndex( )	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous( )	Returns the previous element. A **NoSuchElementException** is thrown if there is not a previous element.
int previousIndex( )	Returns the index of the previous element. If there is not a previous element, returns –1.
void remove( )	Removes the current element from the list. An **IllegalStateException** is thrown if **remove( )** is called before **next( )** or **previous( )** is invoked.
void set(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

Table 18-9 The Methods Provided by **ListIterator**

Using an Iterator

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time. In general, to use an iterator to cycle through the contents of a collection, follow these steps:

1. Obtain an iterator to the start of the collection by calling the collection's iterator( ) method.
2. Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.
3. Within the loop, obtain each element by calling next( ).

For collections that implement List, you can also obtain an iterator by calling listIterator( ). As explained, a list iterator gives you the ability to access the collection in either the forward or backward direction and lets you modify an element. Otherwise, ListIterator is used just like Iterator.

The following example implements these steps, demonstrating both the Iterator and ListIterator interfaces. It uses an ArrayList object, but the general principles apply to any type of collection. Of course, ListIterator is available only to those collections that implement the List interface.

```
// Demonstrate iterators.
import java.util.*;
class IteratorDemo {
 public static void main(String args[]) {
 // Create an array list.
 ArrayList<String> al = new ArrayList<String>();
 // Add elements to the array list.
 al.add("C");
 al.add("A");
 al.add("E");
 al.add("B");
 al.add("D");
 al.add("F");
 // Use iterator to display contents of al.
 System.out.print("Original contents of al: ");
 Iterator<String> itr = al.iterator();
 while(itr.hasNext()) {
 String element = itr.next();
 System.out.print(element + " ");
 }
 System.out.println();
 // Modify objects being iterated.
 ListIterator<String> litr = al.listIterator();
 while(litr.hasNext()) {
 String element = litr.next();
 litr.set(element + "+");
 }
 System.out.print("Modified contents of al: ");
 itr = al.iterator();
 while(itr.hasNext()) {
 String element = itr.next();
 System.out.print(element + " ");
 }
 System.out.println();
 // Now, display the list backwards.
 System.out.print("Modified list backwards: ");
 while(litr.hasPrevious()) {
String element = litr.previous();
 System.out.print(element + " ");
 }
 System.out.println();
 }
}
```

The output is shown here:

```
 Original contents of al: C A E B D F
 Modified contents of al: C+ A+ E+ B+ D+ F+
 Modified list backwards: F+ D+ B+ E+ A+ C+
```

Pay special attention to how the list is displayed in reverse. After the list is modified, litr points to the end of the list. (Remember, litr.hasNext( ) returns false when the end of the list has been reached.) To traverse the list in reverse, the program continues to use litr, but this time it checks to see whether it has a previous element. As long as it does, that element is obtained and displayed.

The For-Each Alternative to Iterators

If you won’t be modifying the contents of a collection or obtaining elements in reverse order, then the for-each version of the for loop is often a more convenient alternative to

cycling through a collection than is using an iterator. Recall that the for can cycle through any collection of objects that implement the Iterable interface. Because all of the collection classes implement this interface, they can all be operated upon by the for.
The following example uses a for loop to sum the contents of a collection:

```
// Use the for-each for loop to cycle through a collection.
import java.util.*;
class ForEachDemo {
 public static void main(String args[]) {
 // Create an array list for integers.
 ArrayList<Integer> vals = new ArrayList<Integer>();
 // Add values to the array list.
 vals.add(1);
 vals.add(2);
 vals.add(3);
 vals.add(4);
 vals.add(5);
 // Use for loop to display the values.
 System.out.print("Contents of vals: ");
 for(int v : vals)
 System.out.print(v + " ");
 System.out.println();
 // Now, sum the values by using a for loop
int sum = 0;
 for(int v : vals)
 sum += v;
 System.out.println("Sum of values: " + sum);
 }
}
```

The output from the program is shown here:

```
 Contents of vals: 1 2 3 4 5
 Sum of values: 15
```

As you can see, the for loop is substantially shorter and simpler to use than the iteratorbased approach. However, it can only be used to cycle through a collection in the forward direction, and you can't modify the contents of the collection.

LinkedList Vs. ArrayList

Both the Java ArrayList and `LinkedList` implements the `List` interface of the `Collections` framework. However, there exists some difference between them.

LinkedList	**ArrayList**
Implements `List`, `Queue`, and `Deque` interfaces.	Implements `List` interface.

Stores 3 values (**previous address, data,** and **next address**) in a single position.	Stores a single value in a single position.
Provides the doubly-linked list implementation.	Provides a resizable array implementation.
Whenever an element is added, `prev` and `next` address are changed.	Whenever an element is added, all elements after that position are shifted.
To access an element, we need to iterate from the beginning to the element.	Can randomly access elements using indexes.

Difference between ArrayList and Vector

ArrayList and Vector both implements List interface and maintains insertion order. However, there are many differences between ArrayList and Vector classes that are given below.

ArrayList	**Vector**
1) ArrayList is **not synchronized**.	Vector is **synchronized**.
2) ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity.	Vector **increments 100%** means doubles the array size if the total number of elements exceeds than its capacity.
3) ArrayList is **not a legacy** class. It is introduced in JDK 1.2.	Vector is a **legacy** class.
4) ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object.
5) ArrayList uses the **Iterator** interface to traverse the elements.	A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements.

ArrayList vs HashSet.

Sr. No.	Key	ArrayList	HashSet
1	Implementation	ArrayList is the implementation of the list interface.	HashSet on the other hand is the implementation of a set interface.
2	Internal implementation	ArrayList internally implements array for its implementation.	HashSet internally uses Hashmap for its implementation.
3	Order of elements	ArrayList maintains the insertion order i.e order of the object in which they are inserted.	HashSet is an unordered collection and doesn't maintain any order.
4	Duplicates	ArrayList allows duplicate values in its collection.	On other hand duplicate elements are not allowed in Hashset.
5	Index performance	ArrayList uses index for its performance i.e its index based one can retrieve object by calling get(index) or remove objects by calling remove(index)	HashSet is completely based on object also it doesn't provide get() method.
6	Null Allowed	Any number of null value can be inserted in arraylist without any restriction.	On other hand Hashset allows only one null value in its collection, after which no null value is allowed to be added.