java.util : The Collections Framework

Java.util Package. It contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

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.

java.util contains a wide array of functionality, it is quite large

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

Some of the interfaces defined by java.util are

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

Here we discuss and examine those members of java.util that are the part of the Collections Framework.

Collection Overview

- The Java Collections Framework standardizes the way in which groups of objects are handled by your programs.
- Prior to the Collections Framework, Java provided ad hoc classes such as Dictionary, Vector, Stack, and Properties to store and manipulate groups of objects.
- The Collections Framework was designed to meet several goals.
- First, the framework had to be high-performance.
- The implementations for the fundamental collections are highly efficient.

- Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- Third, extending and/or adapting a collection had to be easy. The entire Collections Framework is built upon a set of standard interfaces.
- Several standard implementations (such as Linked List, Hash Set, and Tree Set) of these interfaces are also provided.
- You may also implement your own collection, if you choose. Various special-purpose implementations are created.
- Finally, mechanisms were added that allow the integration of standard arrays into the Collections Framework.

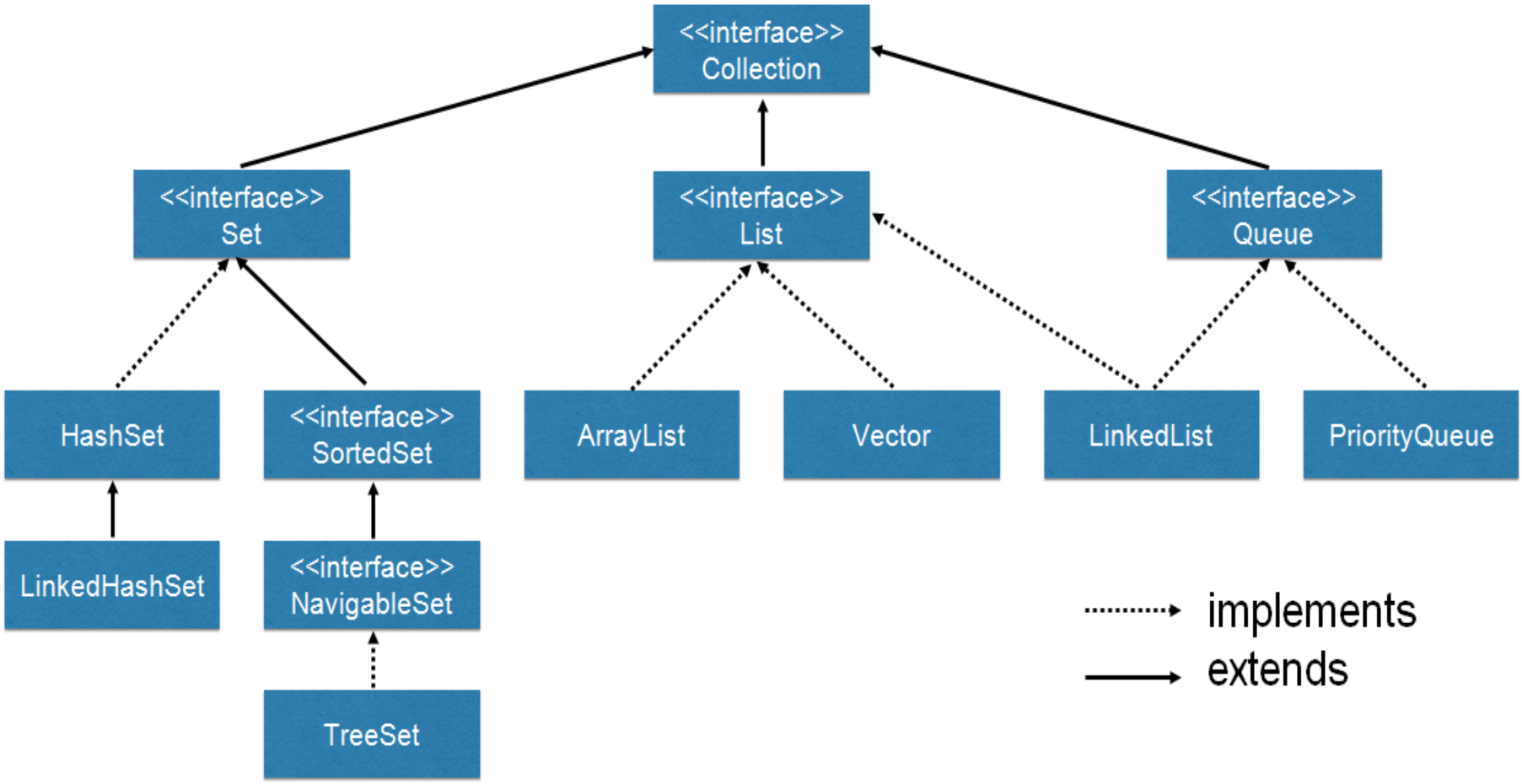
- Algorithms are another important part of the collection mechanism. Algorithms operate on collections and are defined as static methods within the Collections class.
- They provide a standard means of manipulating collections.
- Another item closely associated with the Collections Framework is the Iterator interface.
- An iterator offers a general-purpose, standardized way of accessing the elements within a collection, one at a time.
- Thus, an **iterator** provides a means of enumerating the contents of a collection.

The Collection Interfaces

- The Collections Framework defines several interfaces.
- Collection interfaces is necessary because they determine the fundamental nature of the collection classes.
- Some of the interfaces are:

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

Collection Interface



- In addition to the collection interfaces, collections also use the
  - ✓ **Comparator** : that defines how two objects are compared
  - ✓ **RandomAccess**: By implementing Random Access, a list indicates that it supports efficient, random access to its elements.
  - ✓ **Iterator, and ListIterator interfaces**: enumerate the objects within a collection.

- 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 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.

- Collection declares the core methods that all collections will have.
- Since, all collections implement Collection, familiarity with its methods is necessary for a clear understanding of the framework.
- Several of these methods can throw an **UnsupportedOperationException.**
- this occurs if a collection cannot be modified.
- **AClassCastException** is generated when one **object is incompatible with another**, such as when an attempt is made to add an incompatible object to a collection.

- **A NullPointerException** is thrown if an attempt is made to store a **null object and null elements** are not allowed in the collection.
- **An IllegalArgumentException** is thrown if an invalid argument is used.
- **An IllegalStateException** is thrown if an attempt is made to **add an element to a fixed-length** collection that is full.

- Objects are added to a collection by calling **add( ).**
- Notice that **add( )** takes an argument of **type E**, which means that **objects added to a collection**
- It 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( ).**
- 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.
- One more very important method is **iterator( ),** which returns an iterator to a collection.
- Iterator's are frequently used when working with collections.

Method	Description
boolean add(E *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the operation succeeded (i.e., the elements were added). Otherwise, returns **false**.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the hash code for the invoking collection.
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator<E> iterator( )	Returns an iterator for the invoking collection.
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection<?> *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
boolean retainAll(Collection<?> *c*)	Removes all elements from the invoking collection except those in *c*. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
int size( )	Returns the number of elements held in the invoking collection.
Object[ ] toArray( )	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
<T> T[ ] toArray(T *array*[ ])	Returns an array that contains the elements of the invoking collection. The array elements are copies of the collection elements. If the size of *array* equals the number of elements, these are returned in *array*. If the size of *array* is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of *array* is greater than the number of elements, the array element following the last collection element is set to **null**. An **ArrayStoreException** is thrown if any collection element has a type that is not a subtype of *array*.

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

The List Interface

- The List interface extends Collection and declares the behavior of a collection that stores a sequence of 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.
- List defines some of its own methods.
- These methods will throw an **UnsupportedOperationException**
- if the list cannot be modified, and a **ClassCastException** is generated when one object is incompatible with another.

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

- To obtain the object stored at a specific location, call **get( )** with the index of the object.
- To assign a value to an element in the list, call **set( )**, specifying the index of the object to be changed.
- To find the index of an object, use **indexOf( ) or lastIndexOf( ).**
- 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.

The Set Interface

- The Set interface defines a set.
- It extends Collection and declares the behaviour of a collection that does not allow duplicate elements.
- Therefore, the add( ) method returns false if an attempt is made to add duplicate elements to a set. It does not define any additional methods of its own.
- Set is a generic interface that has this declaration:

  interface Set<E>
- Here, E specifies the type of objects that the set will hold.

The SortedSet Interface

- The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order.
- SortedSet is a generic interface that has this declaration:

  interface SortedSet<E>
- Here, E specifies the type of objects that the set will hold.
- In addition to those methods defined by Set, the SortedSet interface declares the methods

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

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

- Several methods throw a **NoSuchElementException** when no items are contained in the invoking set.
- A **ClassCastException** is thrown when an object is incompatible with the elements in a set.
- A**NullPointerException** is thrown if an attempt is made to use a null object and null is not allowed in the set.
- An **IllegalArgumentException** is thrown if an invalid argument is used.

- **SortedSet** defines several methods that make set processing more convenient.
- To obtain the first object in the set, call **first( ).**
- To get the last element, use **last( ).**
- You can obtain a subset of a sorted set by calling **subSet( )**, specifying the first and last object in the set.
- If you need the subset that starts with the first element in the set, use **headSet( ).**
- If you want the subset that ends the set, use **tailSet( ).**

The NavigableSet Interface

- The NavigableSet interface was added by Java SE 6.
- It extends SortedSet and declares the behaviour of a collection that supports the retrieval of elements based on the closest match to a given value or values.
- NavigableSet is a generic interface that has this declaration:

interface NavigableSet<E>

- Here, E specifies the type of objects that the set will hold.
- In addition to the methods that it inherits from SortedSet

NavigableSet adds those summarized

Method	Description
E ceiling(E *obj*)	Searches the set for the smallest element *e* such that *e* >= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.
NavigableSet<E> descendingSet( )	Returns a **NavigableSet** that is the reverse of the invoking set. The resulting set is backed by the invoking set.
E floor(E *obj*)	Searches the set for the largest element *e* such that *e* <= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
NavigableSet<E> headSet(E *upperBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
E higher(E *obj*)	Searches the set for the largest element *e* such that *e* > *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E lower(E *obj*)	Searches the set for the largest element *e* such that *e* < *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E pollFirst( )	Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. **null** is returned if the set is empty.
E pollLast( )	Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. **null** is returned if the set is empty.
NavigableSet<E> subSet(E *lowerBound*, boolean *lowIncl*, E *upperBound*, boolean *highIncl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
NavigableSet<E> tailSet(E *lowerBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound*. If *incl* is **true**, then an element equal to **lowerBound** is included. The resulting set is backed by the invoking set.

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

The Queue Interface

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

interface Queue<E>

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

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

- Several methods throw a **ClassCastException** when an object is incompatible with the elements in the queue.
- A **NullPointerException** is thrown if an attempt is made to store a null object and null elements are not allowed in the queue.
- An **IllegalArgumentException** is thrown if an invalid argument is used.
- An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length queue that is full.
- A **NoSuchElementException** is thrown if an attempt is made to remove an element from an empty queue.

- Despite its simplicity, Queue offers several points of interest.
- First, **elements can only be removed** from the head of the queue.
- Second, there are two methods that **obtain and remove** elements: **poll( ) and remove( ).**
- The difference between them is that **poll( ) returns null** if the queue is empty, but **remove( ) throws an exception**.
- Third, there are two methods, **element( ) and peek( ),** that obtain but don't remove the element at the head of the queue.
- They differ only in that **element( )** throws an exception if the queue is empty, but **peek( ) returns null**.
- Finally, notice that **offer( )** only attempts to add an element to a queue. Because some queues have a fixed length and might be full, **offer( ) can fail.**

The Deque Interface

- The **Deque** interface was added by **Java SE 6.**
- It extends Queue and declares the behaviour of a double-ended queue.
- Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks.
- Deque is a generic interface that has this declaration:

  interface Deque<E>
- Here, E specifies the type of objects that the deque will hold.
- In addition to the methods that it inherits from Queue, Deque adds those methods summarized

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

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

The Collection Classes

- Now that you are familiar with the collection interfaces
- 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.
- The standard collection classes are summarized

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

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.

- 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.

- ArrayList has the constructors shown here:

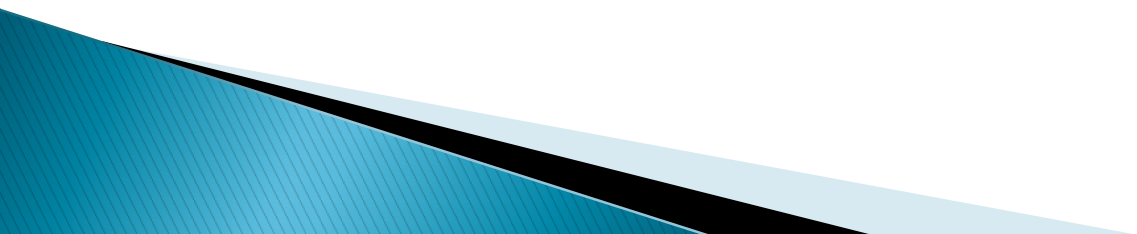
  ArrayList( )

  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.
- The list is then displayed. Some of the elements are removed and the list is displayed again.

Program ArrayListDemo



- Notice that a1 starts out empty and grows as elements are added to it.
- When elements are removed, its size is reduced.
- In the preceding example, the contents of a collection are displayed using the default conversion provided by toString( ), which was inherited from AbstractCollection.
- Although the capacity of an ArrayList object increases automatically as objects are stored in it,
- We can increase the capacity of an ArrayList object manually by calling ensureCapacity( ).

- If it might want to do this if you know in advance that you will be storing many more items in the collection than it can currently hold.
- By increasing its capacity once, at the start, we can prevent several reallocations later.
- Because reallocations are costly in terms of time, preventing unnecessary ones improves performance. The signature for ensureCapacity( ) is shown here:

```
void ensureCapacity(int cap)
```

Here, cap is the new capacity.

- Conversely, if you want to reduce the size of the array that underlies an ArrayList object so that it is precisely as large as the number of items that it is currently holding, call trimToSize( ),

  void trimToSize( )

- **Why ArrayList is better than Array?**

The limitation with **array** is that it has a fixed length so if it is full you cannot add any more elements to it, likewise if there are number of elements gets removed from it the memory consumption would be the same as it doesn't shrink.

On the other **ArrayList** can dynamically grow and shrink after addition and removal of elements. Apart from these benefits ArrayList class enables us to use predefined methods of it which makes our task easy.

How to create an ArrayList?

- We can create an **ArrayList** by writing a simple statement like this:
- This statement creates an **ArrayList** with the name **alist OR aL** with type "String". The type determines which type of elements the list will have. Since this list is of "String" type, the elements that are going to be added to this list will be of type "String".

```
ArrayList<String>alist=new ArrayList<String>( );
```

- Similarly we can create ArrayList that accepts int elements.

```
ArrayList<Integer>list=new ArrayList<Integer>( );
```

How to add elements to an ArrayList?

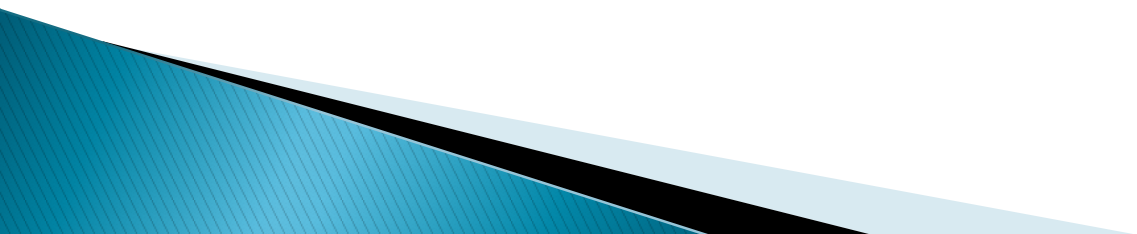
- We add elements to an ArrayList by using **add()** method, this method has couple of variations, which we can use based on the requirement.
- For example: If we want to add the element at the end of the List then simply do it like this:

```
alist.add("Steve"); //This will add "Steve" at the end of List
```

- To add the element at the specified location in ArrayList, we can specify the index in the add method like this:

```
alist.add(3, "Steve"); //This will add "Steve" at the fourth position
```

Program: JavaExample



How to remove elements from ArrayList?

- We use remove() method to remove elements from an ArrayList

Program: JavaExample1
Program:AddRemove

Obtaining an Array from an ArrayList

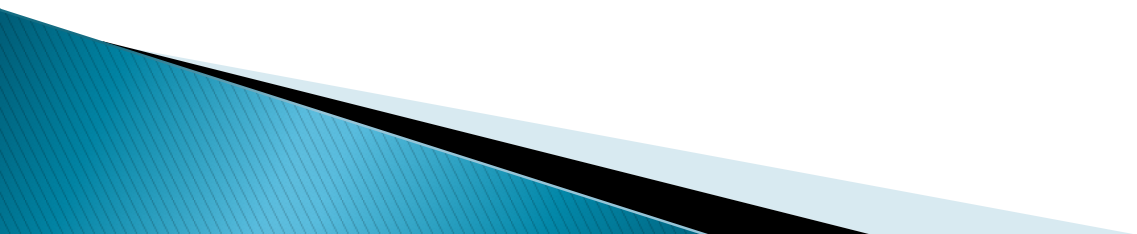
- When working with **ArrayList**, sometimes we want to obtain an actual array that contains the contents of the list.
- We can do this by calling **toArray( ),** which is defined by **Collection.**
- Reasons exist why we might want to convert a collection into an array, such as:
  i. To obtain faster processing times for certain operations
  ii. To pass an array to a method that is not overloaded to accept a collection
  iii. To integrate collection-based code with legacy code that does not understand collections

- There are two versions of toArray( ), which are shown again here for our convenience:

```
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:

Program: ArrayListToArray



- 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.
- There is something else of interest in this program.
- As you know, collections can store only references to, not values of, primitive types.
- However, autoboxing makes it possible to pass values of type int to add( ) without having to manually wrap them within an Integer, as the program shows.
- Autoboxing causes them to be automatically wrapped.
- In this way, autoboxing significantly improves the ease with which collections can be used to store primitive values.

The LinkedList Class

- The LinkedList class extends AbstractSequentialList and implements the List, Deque, and Queue interfaces.
- It provides a linked-list data structure. LinkedList is a generic class that has this declaration:

  class LinkedList<E>

- Here, E specifies the type of objects that the list will hold.
- LinkedList has the two constructors 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.
- 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:

Program: LinkedListDemo

- Because LinkedList implements the List interface, calls to add(E) append items to the end of the list, as do calls to addLast( ).
- To insert items at a specific location, use the add(int, E) form of add( ), as illustrated by the call to add(1, “A2”) in the example.
- Notice how the third element in ll is changed by employing calls to get( ) and set( ).
- To obtain the current value of an element, pass get( ) the index at which the element is stored.
- To assign a new value to that index, pass set( ) the index and its new value.

Other Program Examples

- Program: LinkedList1.java
- Program: LinkedList2.java
- Program: LinkedList3.java
- Program: LinkedList3.java

The HashSet Class

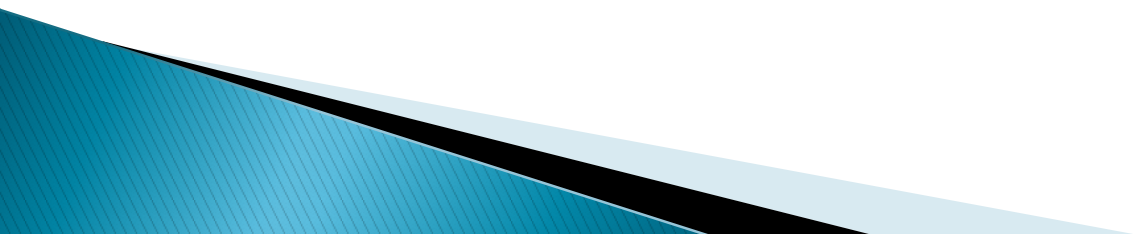
- HashSet extends AbstractSet and implements the Set interface.
- It creates a collection that uses a hash table for storage.
- HashSet is a generic class that has this declaration:

  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.
- Also, your code can't directly index the hash table.
- The advantage of hashing is that it allows the execution time of add( ), contains( ), remove( ), and size( ) to remain constant even for large sets.

Program: HashSetDemo
Program: HashSet2
Program: HashSet3



Accessing a Collection via an Iterator

- An **iterator** provides a means of enumerating the contents of a collection.
- suppose we 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.

- In other words, Iterator is used for iterating (looping) various collection classes such as HashMap, ArrayList, LinkedList etc.
- Iterator took place of Enumeration, which was used to iterate legacy classes such as Vector.
- 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

- 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( ).

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.

Program: IteratorDemo

The For-Each Alternative to Iterators

- If you won't be modifying the contents of a collection or obtaining elements in reverse order,
- 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:

Program: ForEachDemo

As you can see, the for loop is substantially shorter and simpler to use than the iterator based 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.

Storing User-Defined Classes in Collections

- For the sake of simplicity, we can even store the built-in objects, such as Strings or Integers, in a collection.
- The power of collections is that they can store any type of object, including objects of classes that you create.

Program: MailList
Program: MailList1