UNIT-4
Collections Framework(java.util)

The **java.util** package contains a wide array of functionality.

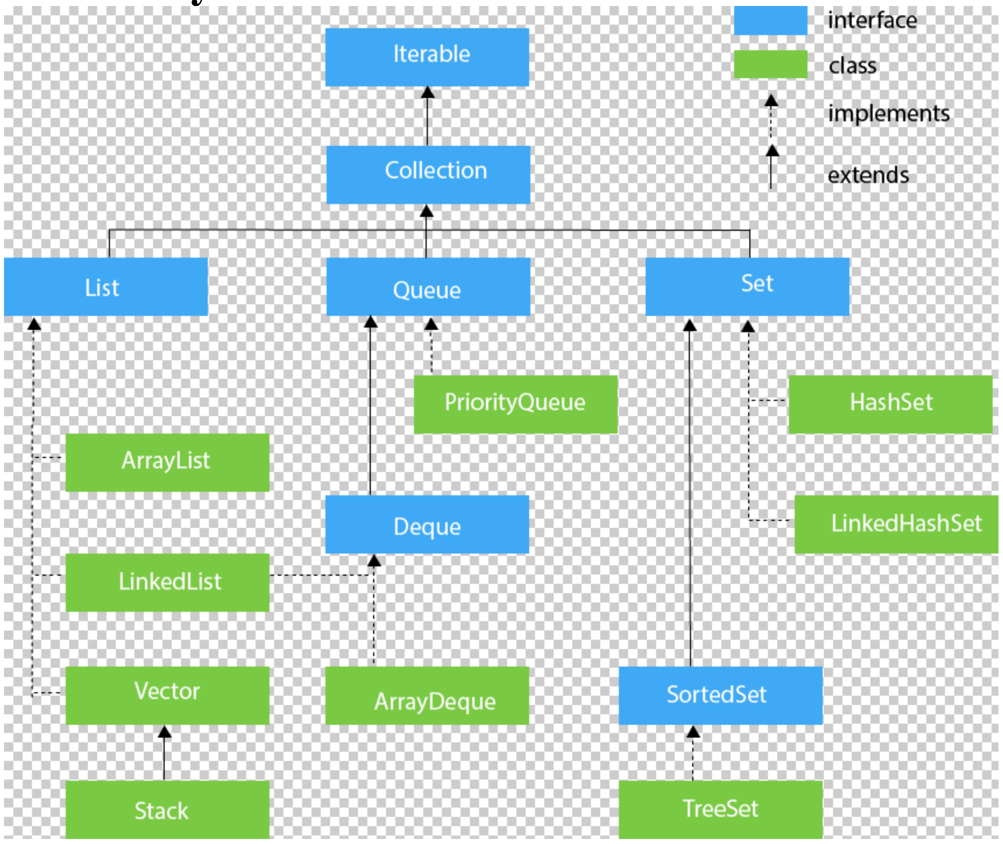
Classes of java.util package:

AbstractCollection	FormattableFlags	Properties
AbstractList	Formatter	PropertyPermission
AbstractMap	GregorianCalendar	PropertyResourceBundle
AbstractQueue	HashMap	Random
AbstractSequentialList	HashSet	ResourceBundle
AbstractSet	Hashtable	Scanner
ArrayDeque	IdentityHashMap	ServiceLoader
ArrayList	IntSummaryStatistics (Added by JDK 8.)	SimpleTimeZone
Arrays	LinkedHashMap	Spliterators (Added by JDK 8.)
Base64 (Added by JDK 8.)	LinkedHashSet	SplitableRandom (Added by JDK 8.)
BitSet	LinkedList	Stack
Calendar	ListResourceBundle	StringJoiner (Added by JDK 8.)
Collections	Locale	StringTokenizer
Currency	LongSummaryStatistics (Added by JDK 8.)	Timer
Date	Objects	TimerTask
Dictionary	Observable	TimeZone
DoubleSummaryStatistics (Added by JDK 8.)	Optional (Added by JDK 8.)	TreeMap
EnumMap	OptionalDouble (Added by JDK 8.)	TreeSet
EnumSet	OptionalInt (Added by JDK 8.)	UUID
EventListenerProxy	OptionalLong (Added by JDK 8.)	Vector
EventObject	PriorityQueue	WeakHashMap

The interfaces defined by **java.util** are shown next:

Collection	Map.Entry	Set
Comparator	NavigableMap	SortedMap
Deque	NavigableSet	SortedSet
Enumeration	Observer	Spliterator (Added by JDK 8.)
EventListener	PrimitiveIterator (Added by JDK 8.)	Spliterator.OfDouble (Added by JDK 8.)
Formattable	PrimitiveIterator.OfDouble (Added by JDK 8.)	Spliterator.OfInt (Added by JDK 8.)
Iterator	PrimitiveIterator.OfInt (Added by JDK 8.)	Spliterator.OfLong (Added by JDK 8.)
List	PrimitiveIterator.OfLong (Added by JDK 8.)	Spliterator.OfPrimitive (Added by JDK 8.)
ListIterator	Queue	
Map	RandomAccess	

Hierarchy of Collection Framework



Collections Overview

The **java.util** package contains the *Collections Framework*.

- The **Collection in Java** is a framework that provides architecture to store and manipulate the group of objects.
- All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion, etc. can be achieved by Java Collections.
- Java Collection means a single unit of objects.
- The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects.
- added by J2SE 1.2.

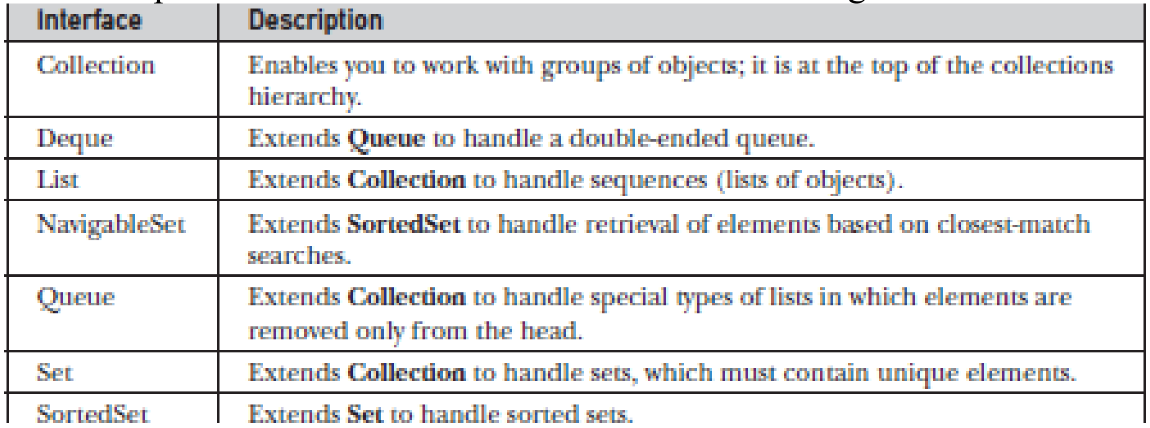
The Collections Framework was designed to meet several goals.

- First, the framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) 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. Toward this end, the entire Collections Framework is built upon a set of standard interfaces. Several standard implementations (such as **LinkedList**, **HashSet**, and **TreeSet**) of these interfaces are provided that you may use as-is. You may also implement your own collection, if you choose. Various special-purpose implementations are created for your convenience, and some partial implementations are provided that make creating your own collection class easier.
- Finally, mechanisms were added that allow the integration of standard arrays into the Collections Framework.

The Collection Interfaces

The Collections Framework defines several core 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.

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 foreach style **for** loop.
- All collection implement **Collection**

Collection methods are summarized in Table

Method	Description
public boolean add(Object element)	is used to insert an element in this collection.
public booleanaddAll(Collection c)	is used to insert the specified collection elements in the invoking collection.
public boolean remove(Object element)	is used to delete an element from this collection.
public booleanremoveAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
public booleanretainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
public int size()	return the total number of elements in the collection.
public void clear()	removes the total no. of elements from the collection.
public boolean contains(Object element)	is used to search an element.
public booleancontainsAll(Collection c)	is used to search the specified collection in this collection.
public Iterator iterator()	returns an iterator.
public Object[] toArray()	converts collection into array.
public booleanisEmpty()	checks if collection is empty.
public boolean equals(Object element)	matches two collections.
public inthashCode()	returns the hash code number of the collection.

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.

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

The List Interface

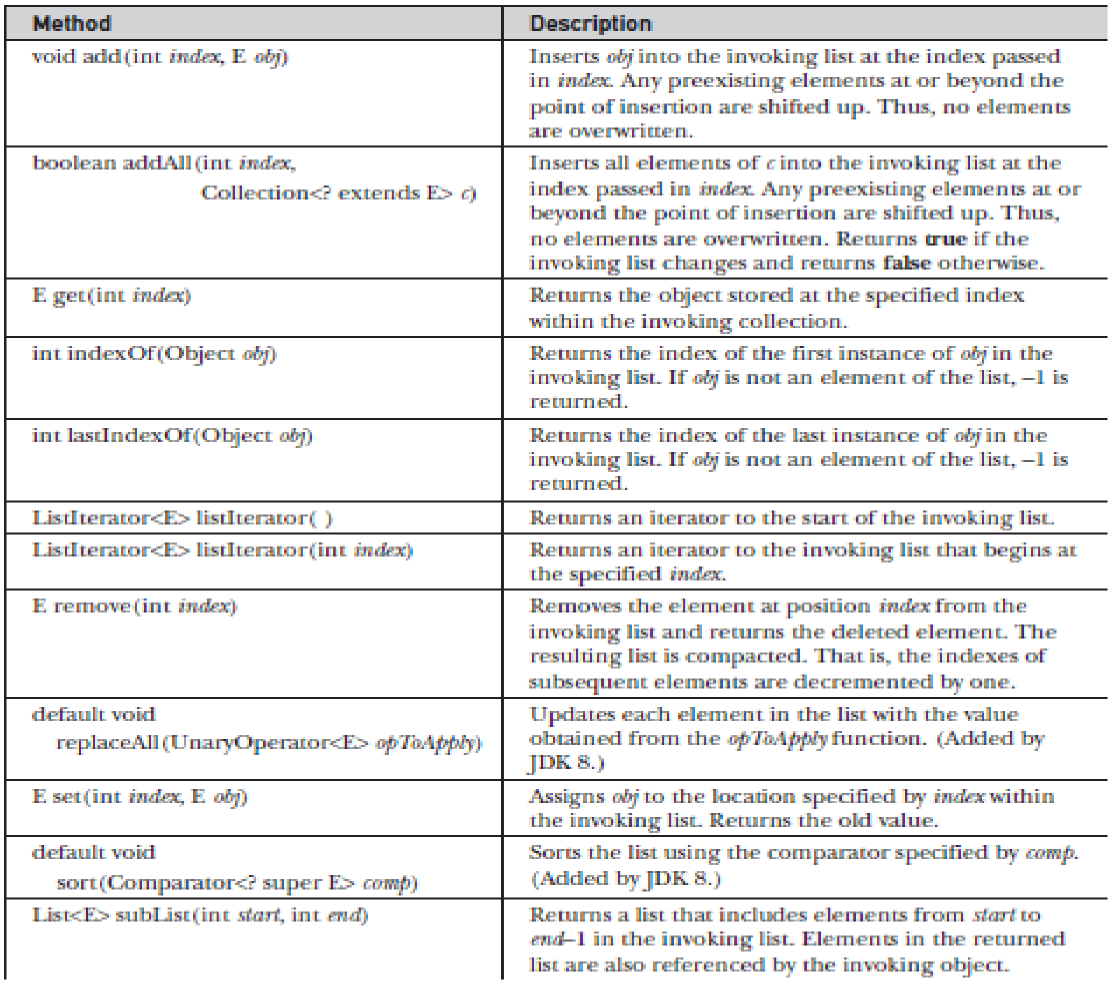
- The **List** interface extends **Collection**
- **It** stores a sequence of elements.
- Elements can be inserted or accessed by their position in the list, using a zero-based index.

- 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



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.

Several of these methods will throw an **UnsupportedOperationException, ClassCastException, IndexOutOfBoundsException, NullPointerException, IllegalArgumentException**

The Set Interface

- The **Set** interface defines a set.
- It extends **Collection**
- **D**oes 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 SortedSet Interface

- The **SortedSet** interface extends **Set**
- It sorts the elements 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 provided by **Set**, the **SortedSet** interface declares the methods summarized in Table

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.

Several of these methods will throw an **NosuchElementException, ClassCastException, NullPointerException, IllegalArgumentException**

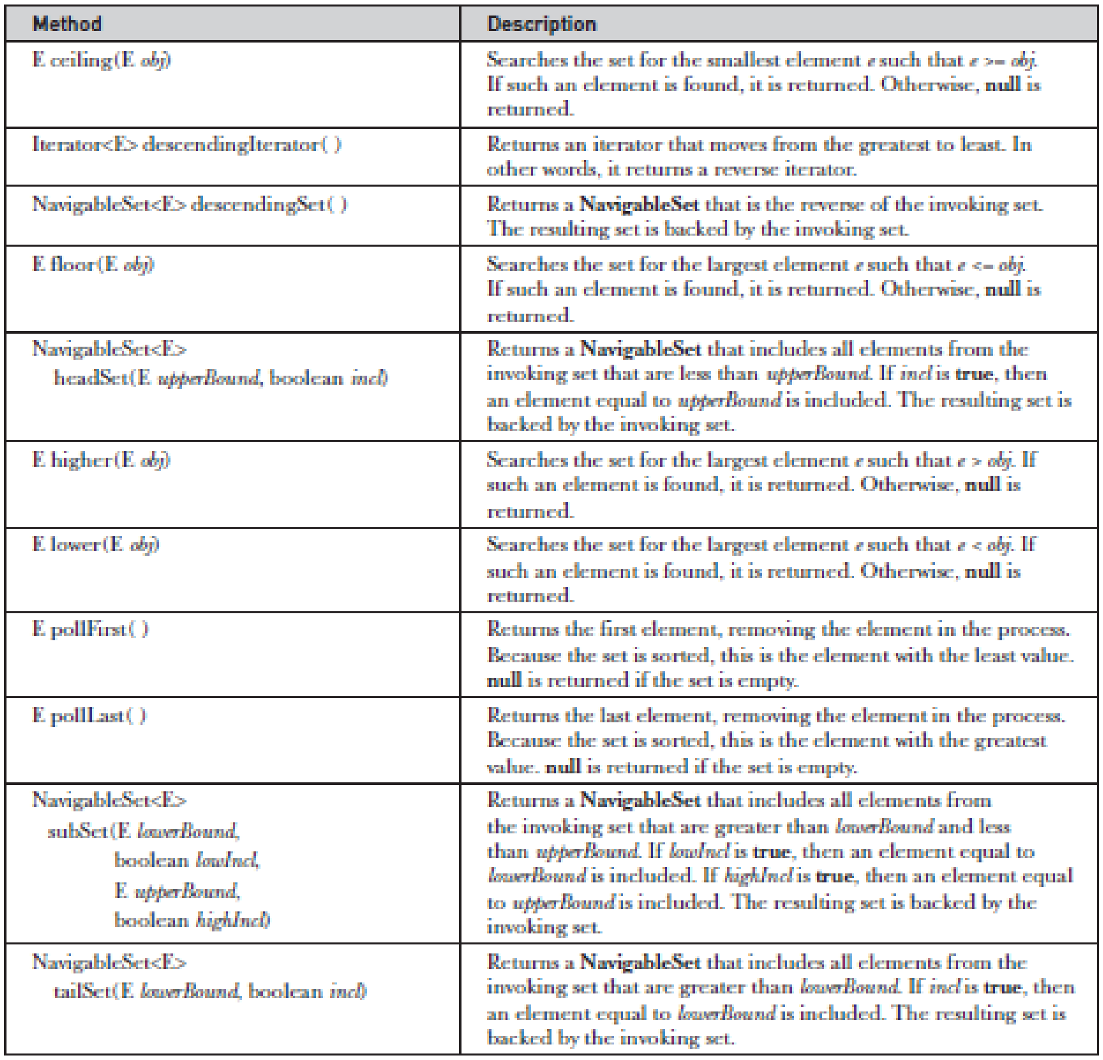
The NavigableSet Interface

- The **NavigableSet** interface extends **SortedSet**
- **It** supports the retrieval of elements based on the closest match to a given value or values.
- **NavigableSet** is a generic interface that has this declaration:

  interface NavigableSet<E>

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

In addition to the methods that it inherits from **SortedSet**, **NavigableSet** adds those summarized in Table



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.

Several of these methods will throw an **ClassCastException, NullPointerException, IllegalArgumentException**

The Queue Interface

- The **Queue** interface extends **Collection**
- **It follows** first-in, first-out list.
- Null elements are not allowed in queue.
- **Queue** is a generic interface that has this declaration:

  interface Queue<E>

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

The methods declared by **Queue** are shown in Table

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.

Several of these methods will throw an **ClassCastException, NullPointerException, IllegalArgumentException, Illegal state exception NoSuchElementException**

The Deque Interface

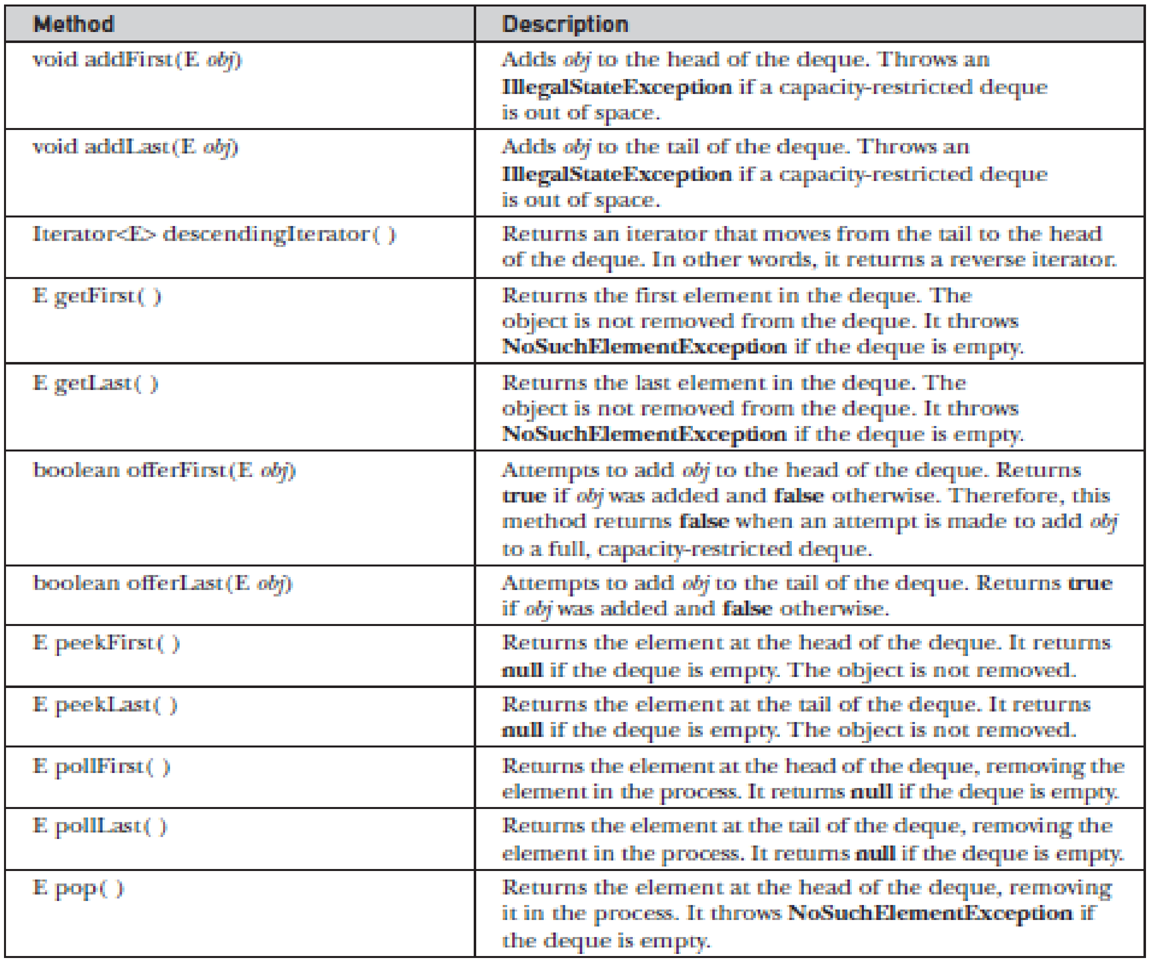
- The **Deque** interface extends **Queue**
- It is like double-ended queue.
- Double-ended queues can function as standard, first-in, first-out queues or as last-in, firstout stacks.
- A **Deque** implementation can be *capacityrestricted*, which means that only a limited number of elements can be added to the deque. When this is the case, an attempt to add an element to the deque can fail.
- **Deque** is a generic interface that has this declaration:

  interface Deque<E>

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

In addition to the methods that it inherits from **Queue**, **Deque** adds those methods summarized in Table

Method	Description
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*.



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.

Several of these methods will throw an **ClassCastException, NullPointerException, IllegalArgumentException, IllegalStateException, NoSuchElement**

The Collection Classes

The core collection classes are summarized in the following table:

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 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(they can grow or shrink based on number of elements).
- 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( ): builds an empty array list
- ArrayList(Collection<? extends E> *c*): builds an array list that is initialized with the elements of the collection c.
- ArrayList(int *capacity*): 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.

```
Program:    // 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(1, "A2");//1 is index in the arraylist
                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(2); // 2 is the index in the array.
                        //we can also write al.remove("A");
                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: 4
Contents of al: [C, A2, A, E]
Size of al after deletions: 3
Contents of al: [C, A2, E]
```

You can increase the size manually by using **ensureCapacity( )**.

void ensureCapacity(int *cap*)

Here, *cap* specifies the new minimum capacity of the collection.

You can decrease the size manually by using **trimToSize( )**

void trimToSize( )

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

There are two versions of **toArray( )**, which are shown:

object[ ] toArray( ) : returns an array of Object
<T> T[ ] toArray(T *array*[ ]): returns an array of elements that have the same type as T.

```
// Convert an ArrayList into an array.
import java.util.*;
class ArrayListDemo
{       public static void main(String args[])
        {       ArrayList<Integer> al = new ArrayList<Integer>();
                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;
                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 LinkedList Class

- LinkedList implements the Collection interface.
- It uses a doubly linked list internally to store the elements.
- It can store the duplicate elements.
- It maintains the insertion order and is not synchronized.
- **LinkedList** is a generic class that has this declaration:

  class LinkedList<E>

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

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

LinkedList has the two constructors shown here:
LinkedList( ): builds an empty linked list.
LinkedList(Collection<? extends E> *c*): builds a linked list that is initialized with the elements of the collection *c*.

```
// Demonstrate LinkedList.
import java.util.*;
class LinkedListDemo
{       public static void main(String args[])
        {       LinkedList<String> ll = new LinkedList<String>();
                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 = ll.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.
- Hashing is used to store the elements in the HashSet. 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.
- It contains the unique items.
- **HashSet** does not define any additional methods beyond those provided by its superclasses and interfaces.
- **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.

The following constructors are defined:

HashSet( ): constructs a default hash set.

HashSet(Collection<?extends E>c):initializes the hash set by using elements of *c*.

HashSet(int *capacity*): initializes the capacity of the hash set to *capacity*. (The default capacity is 16.)

HashSet(int *capacity*, float *fillRatio*): 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.

```
import java.util.*;
class HashSetDemo
{       public static void main(String args[])
        {       HashSet<String> hs = new HashSet<String>();
                hs.add("Beta");
                hs.add("Alpha");
                System.out.println(hs);
        }}The following is the output from this program:   [Alpha, Beta]
```

The TreeSet Class

- Java extends **AbstractSet** and implements the **NavigableSet** interface that uses a tree for storage.
- Like HashSet, TreeSet also contains the unique elements
- However, the access and retrieval time of TreeSet is quite fast.
- The elements in TreeSet stored in ascending order.
- **TreeSet** is a generic class that has this declaration:

  ```
  class TreeSet<E>
  ```

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

TreeSet has the following constructors:

TreeSet( ): constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements.

TreeSet(Collection<? extends E> *c*): builds treeset that contains the elements of *c*.

TreeSet(Comparator<? super E> *comp*): constructs an empty tree set that will be sorted according to the comparator specified by *comp*.

TreeSet(SortedSet<E> *ss*): builds a tree set that contains the elements of *ss*.

```
// Demonstrate TreeSet.
import java.util.*;
class TreeSetDemo
{       public static void main(String args[])
        {       TreeSet<String> ts = new TreeSet<String>();
                ts.add("C");
                ts.add("A");
                ts.add("B");
                ts.add("F");
                ts.add("D");
                System.out.println(ts);
                System.out.println(ts.subSet("C", "F")); //display subset of elements
        }
}
```

Output:

```
[A, B, C, D, F]
[C, D, E]
```

The PriorityQueue Class

- **PriorityQueue** extends **AbstractQueue** and implements the **Queue** interface.
- It creates a queue that is prioritized based on the queue's comparator.
- PriorityQueues are dynamic, growing as necessary.
- PriorityQueue is a generic class that has this declaration:

  class PriorityQueue<E>

  Here, E specifies the type of objects stored in the queue.

PriorityQueue defines the six constructors shown here:
PriorityQueue( ): builds an empty queue. Its starting capacity is 11.
PriorityQueue(int *capacity*): builds a queue that has the specified initial capacity.
PriorityQueue(Comparator<? super E> *comp*) (Added by JDK 8.): specifies a comparator
PriorityQueue(int *capacity*, Comparator<? super E> *comp*): builds a queue with the specified capacity and comparator
PriorityQueue(Collection<? extends E> *c*)
PriorityQueue(PriorityQueue<? extends E> *c*)
PriorityQueue(SortedSet<? extends E> *c*)

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

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

```
import java.util.*;
class TestCollection12
{       public static void main(String args[ ])
        {       PriorityQueue<String> queue=new PriorityQueue<String>( );
                queue.add("Amit");
                queue.add("Vijay");
                queue.add("Karan");
                queue.add("Jai");
                queue.add("Rahul");
                System.out.println("head:"+queue.element( ));
                System.out.println("head:"+queue.peek( ));
                System.out.println("iterating the queue elements:");
                Iterator itr=queue.iterator( );
                while(itr.hasNext( ))
                {       System.out.println(itr.next( ));
                }
                queue.remove( );
                queue.poll( );
                System.out.println("after removing two elements:");
                Iterator<String> itr2=queue.iterator( );
                while(itr2.hasNext( ))
                {       System.out.println(itr2.next( ));
                }
        }
}
```

Output:

```
head:Amit
head:Amit
iterating the queue elements:
Amit
Jai
Karan
Vijay
Rahul
after removing two elements:
Karan
Rahul
Vijay
```

The ArrayDeque Class

- The **ArrayDeque** class extends **AbstractCollection** and implements the **Deque** interface.
- The ArrayDeque class provides the facility of using deque and resizable-array.
- Unlike Queue, we can add or remove elements from both sides.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.
- It adds no methods of its own.
- ArrayDeque creates a dynamic array and has no capacity restriction
- ArrayDeque is faster than LinkedList and Stack.
- ArrayDeque is a generic class that has this declaration:

  ```
  class ArrayDeque<E>
  ```

  Here, E specifies the type of objects stored in the collection.

ArrayDeque defines the following constructors:

ArrayDeque( ): builds an empty deque. Its starting capacity is 16

ArrayDeque(int *size*): builds a deque that has the specified initial capacity
ArrayDeque(Collection<? extends E> *c*): creates a deque that is initialized with the elements of the collection passed in *c*.

```
import java.util.*;
class ArrayDequeDemo
{       public static void main(String args[])
        {       ArrayDeque<String> adq = new ArrayDeque<String>();
                // Use an ArrayDeque like a stack.
                        adq.push("A");
                        adq.push("B");
                        adq.push("D");
                        adq.push("E");
                        adq.push("F");
                System.out.print("Popping the stack: ");
                while(adq.peek() != null)
                System.out.print(adq.pop() + " ");
                System.out.println();
        }
}
```

The output is shown here:

```
Popping the stack: F E D B A
```

Accessing a Collection via an Iterator

- To cycle through the elements an Iterator is used, which is an object that implements either the **Iterator** or the **ListIterator** interface.
- **ListIterator** extends **Iterator** to allow bidirectional traversal of a list, and the modification of elements.

The **Iterator** interface declares the methods shown in Table

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**.

The methods declared by **ListIterator** (along with those inherited from **Iterator**) are shown in Table.

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( )**.

Using an Iterator

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. //Iterator i=hs.iterator()
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( )**.

ListIterator is used just like **Iterator**.

```
import java.util.*;
class IteratorDemo
{       public static void main(String args[])
        {       ArrayList<String> al = new ArrayList<String>();
                al.add("C");
                al.add("A");
                al.add("E");
                // 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
Modified contents of al: C+ A+ E+
Modified list backwards: E+ A+ C+
```

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.
- **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**.

```
// Use the for-each for loop to cycle through a collection.
import java.util.*;
class ForEachDemo
{       public static void main(String args[])
        {       ArrayList<Integer> vals = new ArrayList<Integer>();
                vals.add(1);
                vals.add(2);
                // 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
Sum of values: 3
```

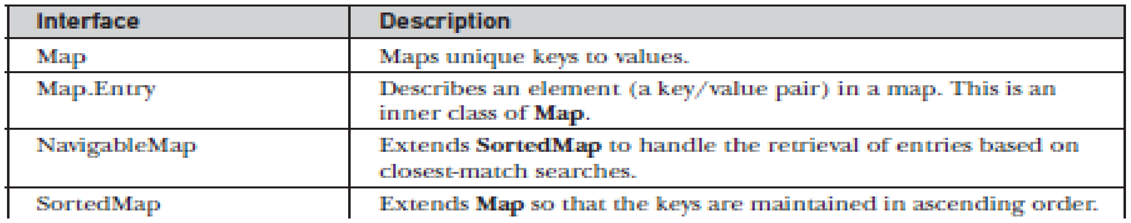
Working with Maps

- A *map* is an object that stores associations between keys and values, or *key/value pairs*.
- Given a key, you can find its value.
- Both keys and values are objects.
- The keys must be unique, but the values may be duplicated.
- Some maps can accept a **null** key and **null** values, others cannot.
- They don't implement the **Iterable** interface.

The Map Interfaces

- Map interfaces define the character and nature of maps

The following interfaces support maps:



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

Several methods throw a **ClassCastException, NullPointerException, UnsupportedOperationException, IllegalArgumentException.**

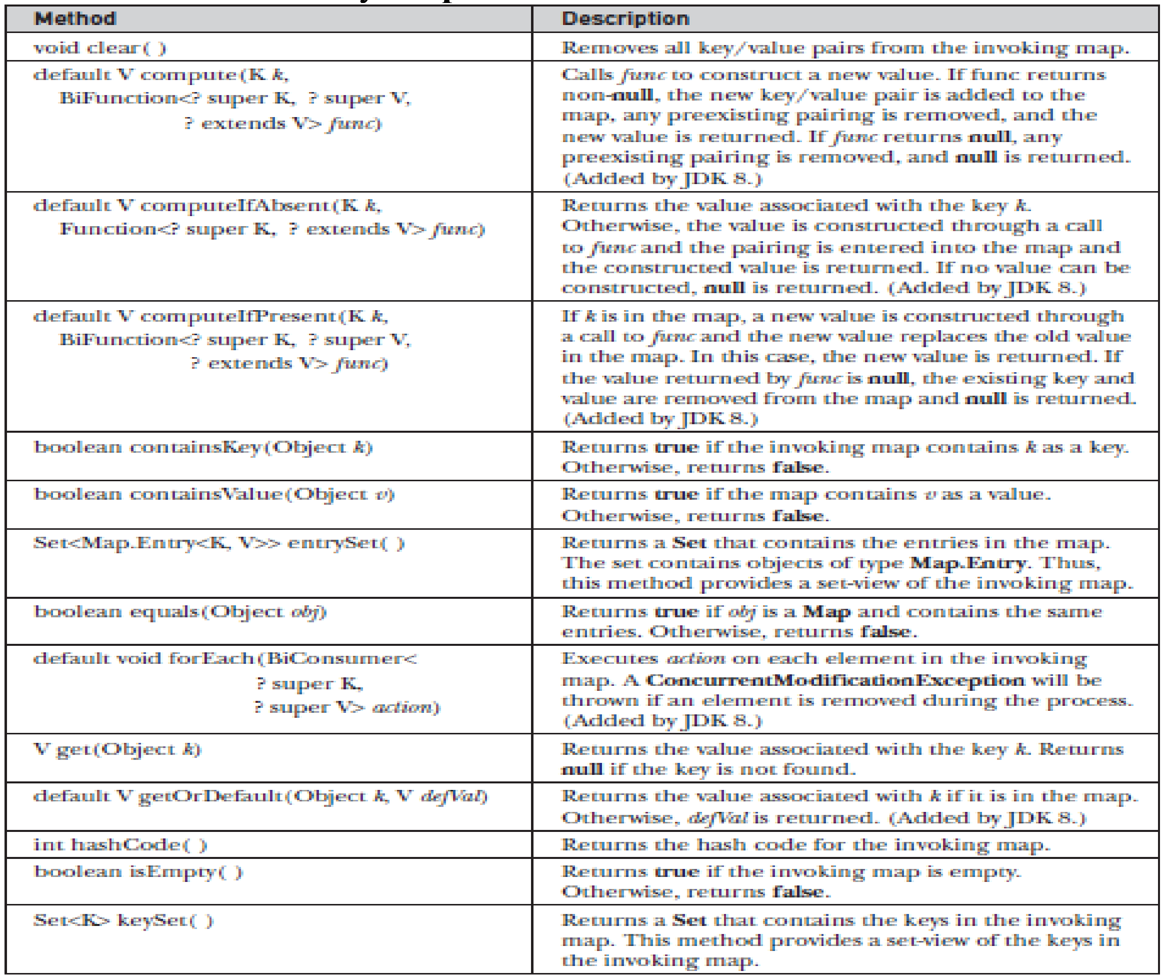
The Map Interface

- The **Map** interface maps unique keys to values.
- A *key* is an object that you use to retrieve a value at a later date.
- Given a key and a value, you can store the value in a **Map** object.
- After the value is stored, you can retrieve it by using its key.
- Although part of the Collections Framework, maps are not, themselves, collections because they do not implement the **Collection** interface. However, you can obtain a collection-view of a map. To do this, you can use the **entrySet( )** method. It returns a **Set** that contains the elements in the map. To obtain a collection-view of the keys, use **keySet( )**. To get a collection-view of the values, use **values( )**. For all three collection views, the collection is backed by the map. Changing one affects the other. Collection-views are the means by which maps are integrated into the larger Collections Framework.
- **Map** is generic and is declared as shown here:

  interface Map<K, V>

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

The methods declared by **Map** are summarized in Table.



Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
default V compute(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	Calls *func* to construct a new value. If func returns non-**null**, the new key/value pair is added to the map, any preexisting pairing is removed, and the new value is returned. If *func* returns **null**, any preexisting pairing is removed, and **null** is returned. (Added by JDK 8.)
default V computeIfAbsent(K *k*, Function<? super K, ? extends V> *func*)	Returns the value associated with the key *k*. Otherwise, the value is constructed through a call to *func* and the pairing is entered into the map and the constructed value is returned. If no value can be constructed, **null** is returned. (Added by JDK 8.)
default V computeIfPresent(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	If *k* is in the map, a new value is constructed through a call to *func* and the new value replaces the old value in the map. In this case, the new value is returned. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned. (Added by JDK 8.)
boolean containsKey(Object *k*)	Returns **true** if the invoking map contains *k* as a key. Otherwise, returns **false**.
boolean containsValue(Object *v*)	Returns **true** if the map contains *v* as a value. Otherwise, returns **false**.
Set<Map.Entry<K, V>> entrySet( )	Returns a **Set** that contains the entries in the map. The set contains objects of type **Map.Entry**. Thus, this method provides a set-view of the invoking map.
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map** and contains the same entries. Otherwise, returns **false**.
default void forEach(BiConsumer< ? super K, ? super V> *action*)	Executes *action* on each element in the invoking map. A **ConcurrentModificationException** will be thrown if an element is removed during the process. (Added by JDK 8.)
V get(Object *k*)	Returns the value associated with the key *k*. Returns **null** if the key is not found.
default V getOrDefault(Object *k*, V *defVal*)	Returns the value associated with *k* if it is in the map. Otherwise, *defVal* is returned. (Added by JDK 8.)
int hashCode( )	Returns the hash code for the invoking map.
boolean isEmpty( )	Returns **true** if the invoking map is empty. Otherwise, returns **false**.
Set<K> keySet( )	Returns a **Set** that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.

Method	Description
default V merge(K *k*, V *v*, BiFunction<? super V, ? super V, ? extends V> *func*)	If *k* is not in the map, the pairing *k,v* is added to the map. In this case, *v* is returned. Otherwise, *func* returns a new value based on the old value, the key is updated to use this value, and **merge( )** returns this value. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned. (Added by JDK 8.)
V put(K *k*, V *v*)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are *k* and *v*, respectively. Returns **null** if the key did not already exist. Otherwise, the previous value linked to the key is returned.
void putAll(Map<? extends K, ? extends V> *m*)	Puts all the entries from *m* into this map.
default V putIfAbsent(K *k*, V *v*)	Inserts the key/value pair into the invoking map if this pairing is not already present or if the existing value is **null**. Returns the old value. The **null** value is returned when no previous mapping exists, or the value is **null**. (Added by JDK 8.)
V remove(Object *k*)	Removes the entry whose key equals *k*.
default boolean remove(Object *k*, Object *v*)	If the key/value pair specified by *k* and *v* is in the invoking map, it is removed and **true** is returned. Otherwise, **false** is returned. (Added by JDK 8.)
default boolean replace(K *k*, V *oldV*, V *newV*)	If the key/value pair specified by *k* and *oldV* is in the invoking map, the value is replaced by *newV* and **true** is returned. Otherwise **false** is returned. (Added by JDK 8.)
default V replace(K *k*, V *v*)	If the key specified by *k* is in the invoking map, its value is set to *v* and the previous value is returned. Otherwise, **null** is returned. (Added by JDK 8.)
default void replaceAll(BiFunction< ? super K, ? super V, ? extends V> *func*)	Executes *func* on each element of the invoking map, replacing the element with the result returned by *func*. A **ConcurrentModificationException** will be thrown if an element is removed during the process. (Added by JDK 8.)
int size( )	Returns the number of key/value pairs in the map.
Collection<V> values( )	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

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

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

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

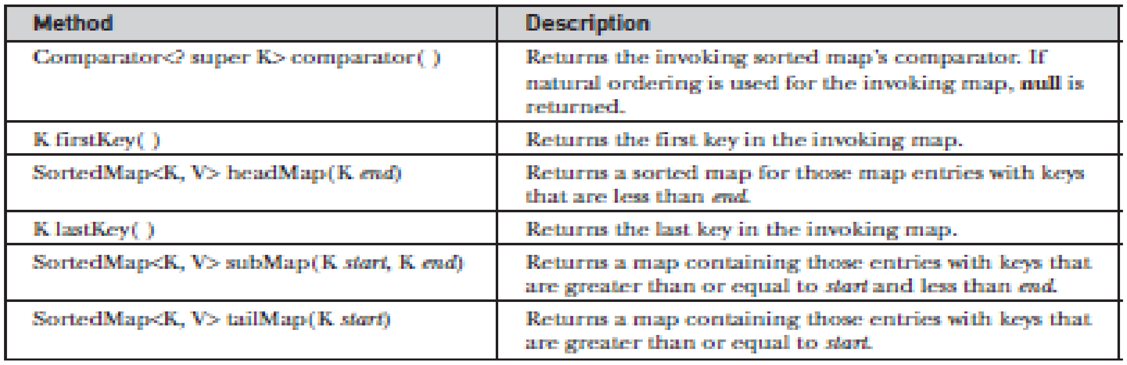
The SortedMap Interface

- The **SortedMap** interface extends **Map**.
- It ensures that the entries are maintained in ascending order based on the keys.
- **SortedMap** is generic and is declared as shown here:

  interface SortedMap<K, V>

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

The methods declared by **SortedMap** are summarized in Table.



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

Several methods throw **NoSuchElementException, ClassCastException, NullPointerException, IllegalArgumentException**

Sorted maps allow very efficient manipulations of *submaps* (in other words, subsets of a map). To obtain a submap, use **headMap( )**, **tailMap( )**, or **subMap()**. The submap returned by these methods is backed by the invoking map. Changing one changes the other. To get the first key in the set, call **firstKey( )**.

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

The NavigableMap Interface

- The **NavigableMap** interface extends **SortedMap** and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys.
- **NavigableMap** is a generic interface that has this declaration:

  interface NavigableMap<K,V>

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

In addition to the methods that it inherits from **SortedMap**, **NavigableMap** adds those summarized in Table

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

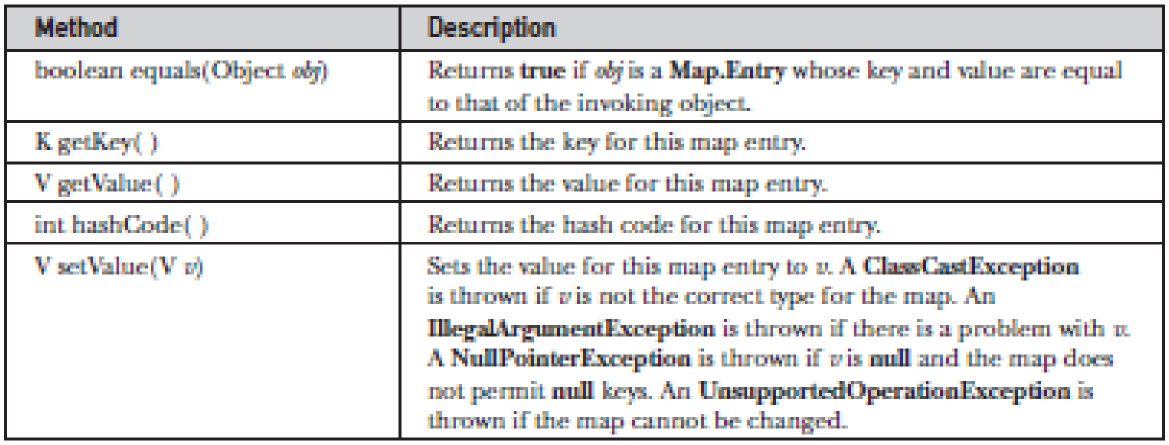
The Map.Entry Interface

- The **Map.Entry** interface enables you to work with a map entry.
- **Map.Entry** is generic and is declared like this:

  interface Map.Entry<K, V>

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

Table summarizes the non-static methods declared by **Map.Entry**.



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

JDK 8 adds two static methods.

- The first is **comparingByKey( )**: returns a **Comparator** that compares entries by key.
- The second is **comparingByValue( ):** returns a **Comparator** that compares entries by value.

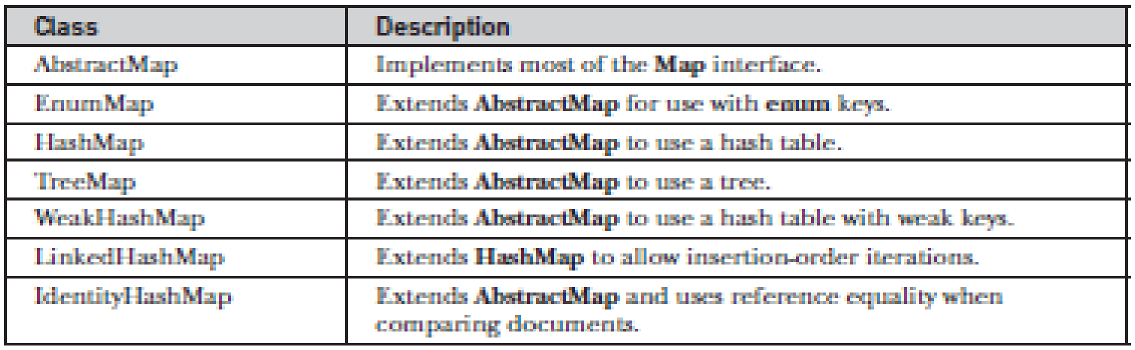
```
import java.util.*;
class MapInterfaceExample
{       public static void main(String args[ ])
        {       Map<Integer,String> map=new HashMap<Integer,String>( );
                map.put(100,"Amit");
                map.put(101,"Vijay");
                map.put(102,"Rahul");
                for(Map.Entry m:map.entrySet( ))
                {       System.out.println(m.getKey( )+" "+m.getValue( ));
                }
        }
}
```

Output:

```
100 Amit
101 Vijay
102 Rahul
```

The Map Classes

Several classes provide implementations of the map interfaces. The classes that can be used for maps are summarized here:



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

The HashMap Class

- The **HashMap** class extends **AbstractMap** and implements the **Map** interface.
- It uses a hash table to store the map.
- This allows the execution time of **get( )** and **put( )** to remain constant even for large sets
- It does not add any methods of its own.
- A hash map does not guarantee the order of its elements.
- **HashMap** is a generic class that has this declaration:

  class HashMap<K, V>

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

The following constructors are defined:

HashMap( ): constructs a default hash map.

HashMap(Map<? extends K, ? extends V> *m*): initializes the hash map by using the elements of *m*.

HashMap(int *capacity*): initializes the capacity of the hash map to *capacity*.

HashMap(int *capacity*, float *fillRatio*): initializes both the capacity and fill ratio of the hash map by using its arguments. The default capacity is 16. The default fill ratio is 0.75.

```
import java.util.*;
class HashMapDemp
{       public static void main(String args[])
        {       HashMap<String, Double> hm = new HashMap<String, Double>();
                // Put elements to the map
                hm.put("John Doe", new Double(3434.34));
                hm.put("Tod Hall", new Double(99.22));
                hm.put("Ralph Smith", new Double(-19.08));
            // Get a set of the entries.
                Set<Map.Entry<String, Double>> set = hm.entrySet();
            // Display the set.
                for(Map.Entry<String, Double> me : set)
                {       System.out.print(me.getKey() + ": ");
                        System.out.println(me.getValue());
                }
                System.out.println();
            // Deposit 1000 into John Doe's account.
                double balance = hm.get("John Doe");
                hm.put("John Doe", balance + 1000);
                System.out.println("John Doe's new balance: " +hm.get("John
                      Doe"));
        }
}
```

Output from this program is shown here (the precise order may vary):

```
Tod Hall: 99.22
John Doe: 3434.34
Ralph Smith: -19.08
John Doe's new balance: 4434.34
```

The TreeMap Class

- The **TreeMap** class extends **AbstractMap** and implements the **NavigableMap** interface.
- It creates maps stored in a tree structure.
- A **TreeMap** provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval.
- It does not have its own methods
- **TreeMap** is a generic class that has this declaration:

  ```
  class TreeMap<K, V>
  ```

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

The following **TreeMap** constructors are defined:

TreeMap( ): constructs an empty tree map that will be sorted by using the natural order of its keys.

TreeMap(Comparator<? super K> *comp*): constructs an empty tree-based map that will be sorted by using the **Comparator** *comp*.

TreeMap(Map<? extends K, ? extends V> *m*): initializes a tree map with the entries from *m*, which will be sorted by using the natural order of the keys.
TreeMap(SortedMap<K, ? extends V> *sm*): initializes a tree map with the entries from *sm*, which will be sorted in the same order as *sm*.

```
import java.util.*;
class TreeMapDemo
{	public static void main(String args[])
	{	TreeMap<String, Double> tm = new TreeMap<String, Double>();
		// Put elements to the map.
			tm.put("John Doe", new Double(3434.34));
			tm.put("Tod Hall", new Double(99.22));
			tm.put("Ralph Smith", new Double(-19.08));
		// Get a set of the entries.
			Set<Map.Entry<String, Double>> set = tm.entrySet();
		// Display the elements.
			for(Map.Entry<String, Double> me : set)
			{	System.out.print(me.getKey() + ": ");
				System.out.println(me.getValue());
			}
			System.out.println();
		// Deposit 1000 into John Doe's account.
			double balance = tm.get("John Doe");
			tm.put("John Doe", balance + 1000);
			System.out.println("John Doe's new balance: " +
			tm.get("John Doe"));
	}
}
```

The following is the output from this program:

```
John Doe: 3434.34
Ralph Smith: -19.08
Tod Hall: 99.22
John Doe's new balance: 4434.34
```

The LinkedHashMap Class

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

- You can also create a **LinkedHashMap** that returns its elements in the order in which they were last accessed.
- **LinkedHashMap** is a generic class that has this declaration:

  class LinkedHashMap<K, V>

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

LinkedHashMap defines the following constructors:

LinkedHashMap( ): constructs a default LinkedHashMap.

LinkedHashMap(Map<? extends K, ? extends V> *m*): initializes the LinkedHashMap with the elements from *m*.

LinkedHashMap(int *capacity*): initializes the capacity.

LinkedHashMap(int *capacity*, float *fillRatio*): initializes both capacity and fill ratio. The meaning of capacity and fill ratio are the same as for HashMap. The default capacity is 16. The default ratio is 0.75.

LinkedHashMap(int *capacity*, float *fillRatio*, boolean *Order*): allows you to specify whether the elements will be stored in the linked list by insertion order, or by order of last access. If *Order* is true, then access order is used. If *Order* is false, then insertion order is used.

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

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

This method is called by **put( )** and **putAll( )**. The oldest entry is passed in *e*. By default, this method returns false and does nothing. However, if you override this method, then you can have the LinkedHashMap remove the oldest entry in the map. To do this, have your override return true. To keep the oldest entry, return false.

The IdentityHashMap Class

- **IdentityHashMap** extends **AbstractMap** and implements the **Map** interface.
- It is similar to **HashMap** except that it uses reference equality when comparing elements
- The API documentation explicitly states that **IdentityHashMap** is not for general use.
- **IdentityHashMap** is a generic class that has this declaration:

  class IdentityHashMap<K, V>

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

The EnumMap Class

- **EnumMap** extends **AbstractMap** and implements **Map**.
- It is specifically for use with keys of an **enum** type.
- **EnumMap** defines no methods of its own.
- It is a generic class that has this declaration:

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

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

EnumMap defines the following constructors:

EnumMap(Class<K> *kType*): creates an empty EnumMap of type *kType*.

EnumMap(Map<K, ? extends V> *m*): creates an EnumMap map that contains the same entries as *m*.

EnumMap(EnumMap<K, ? extends V> *em*): creates an EnumMap initialized with the values in *em*.

Comparators

- Both **TreeSet** and **TreeMap** store elements in sorted order.
- However, it is the comparator that defines precisely what "sorted order" means.
- If you want to order elements a different way, then specify a Comparator when you construct the set or map.
- Comparator is a generic interface that has this declaration:

  interface Comparator<T>

  Here, T specifies the type of objects being compared.

- Prior to JDK 8, the Comparator interface defined only two methods:
  - The **compare( )**: compares two elements for order:

    int compare(T *obj1*, T *obj2*) //throw a **ClassCastException** if the //types of the objects are not compatible for comparison.

    obj1 and *obj2:* objects to be compared.

    if *obj1==obj2 r*eturns zero
    if *obj1 > obj2* returns a positive value
    if *obj1 < obj2* return*s* a negative value.

  - The **equals( ):** tests whether an object equals the invoking comparator

    boolean equals(object *obj*)

    obj is the object to be tested for equality. The method returns true if *obj* and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false.

- JDK 8 adds significant new functionality to Comparator using default and static interface methods.
  - **reversed( ):** reverses the ordering of the comparator on which it is called by using:

    default Comparator<T> reversed( )

    It returns the reverse comparator. For example, assuming a comparator that uses natural ordering for the characters A through Z, a reverse order comparator would put B before A, C before B, so on.

  - **reverseOrder( )**:

  static <T extends Comparable<? super T>> Comparator<T> reverseOrder( )
    It returns a comparator that reverses the natural order of the elements.

  - **naturalOrder( ):** you can obtain a comparator that uses natural ordering by calling the static method, shown next:

  static <T extends Comparable<? super T>> Comparator<T> naturalOrder( )

- If you want a comparator that can handle **null** values, use
  - The **nullsFirst( ):** returns a comparator that views null values as less than other values.
    static <T> Comparator<T> nullsFirst(Comparator<? super T> *comp*)
  - The **nullsLast( ):** returns a comparator that views null values as greater than other values.
    static <T> Comparator<T> nullsLast(Comparator<? super T> *comp*)
    In both cases, if the two values being compared are non-null, *comp* performs the comparison. If *comp* is passed null, then all non-null values are viewed as equivalent.

- **thenComparing( ):** returns a comparator that performs a second comparison when the outcome of the first comparison indicates that the objects being compared are equal. Thus, it can be used to create a "compare by X then compare by Y" sequence. The **thenComparing()** method has three forms.

default Comparator<T> thenComparing(Comparator<? super T> *thenByComp*)

Here, *thenByComp* specifies the comparator that is called if the first comparison returns equal.

The next versions of **thenComparing( )** let you specify the standard functional interface Function (defined by java.util.function). They are:

default <U extends Comparable<? super U> Comparator<T> thenComparing(Function<? super T, ? extends U> *getKey*)

default <U> Comparator<T> thenComparing(Function<? super T, ? extends U> *getKey*) Comparator<? super U> *keyComp*)

In both, *getKey* refers to function that obtains the next comparison key, which is used if the first comparison returns equal. In the second version, *keyComp* specifies the comparator used to compare keys. (Here, and in subsequent uses, **U** specifies the type of the key.)

Comparator also adds the following specialized versions of "then comparing" methods for the primitive types:

default Comparator<T>thenComparingDouble(ToDoubleFunction<? super T> *getKey*)

default Comparator<T> thenComparingInt(ToIntFunction<? super T> *getKey*)

default Comparator<T>thenComparingLong(ToLongFunction<? super T> *getKey*)

In all methods, *getKey* refers to a function that obtains the next comparison key.

- **comparing( )**. It returns a comparator that obtains its comparison key from a function passed to the method.
  There are two versions of **comparing( )**, shown here:

static <T, U extends Comparable<? super U>> Comparator<T>
comparing(Function<? super T, ? extends U> *getKey*)

static <T, U> Comparator<T> comparing(Function<? super T, ? extends U> *getKey*, Comparator<? super U> *keyComp*)

In both, *getKey* refers to a function that obtains the next comparison key. In the second version, *keyComp* specifies the comparator used to compare keys.

Comparator also adds the following specialized versions of these methods for the primitive types:

static <T> Comparator<T> ComparingDouble(ToDoubleFunction<? super T> *getKey*)

static <T> Comparator<T> ComparingInt(ToIntFunction<? super T> *getKey*)

static <T> Comparator<T> ComparingLong(ToLongFunction<? super T> *getKey*)

In all methods, *getKey* refers to a function that obtains the next comparison key.

Using a Comparator

```
// Use a custom comparator.
import java.util.*;
// A reverse comparator for strings.
class MyComp implements Comparator<String>
{       public int compare(String aStr, String bStr)
        {       // Reverse the comparison.
                        return bStr.compareTo(aStr);
        }
        // No need to override equals or the default methods.
}
class CompDemo
{       public static void main(String args[])
        {       // Create a tree set.
                        TreeSet<String> ts = new TreeSet<String>(new MyComp( ));
                // Add elements to the tree set.
                        ts.add("C");
                        ts.add("A");
                        ts.add("B");
                        ts.add("E");
                        ts.add("F");
                        ts.add("D");
                // Display the elements.
                        for(String element : ts)
                                System.out.print(element + " ");
                        System.out.println();
        }
}
```

As the following output shows, the tree is now sorted in reverse order:

```
F E D C B A
```

Beginning with JDK 8, there is another way to approach a solution. It is now possible to simply call **reversed( )** on a natural-order comparator. It will return an equivalent comparator, except that it runs in reverse. For example, you can rewrite **MyComp** as a natural-order comparator, as shown here:

```
        class MyComp implements Comparator<String>
        {       public int compare(String aStr, String bStr)
                {       return aStr.compareTo(bStr);
                }
        }
```

Next, you can use the following sequence to create a **TreeSet** that orders its string elements in reverse:

```
                        MyComp mc = new MyComp(); // Create a comparator
                        // Pass a reverse order version of MyComp to TreeSet.
                    TreeSet<String> ts = new TreeSet<String>(mc.reversed());
```

If you plug this new code into the preceding program, it will produce the same results as before.

Beginning with JDK 8, it is not actually necessary to create the **MyComp** class in the preceding examples because a lambda expression can be easily used instead.

// Use a **lambda expression** to create a reverse comparator.

```
import java.util.*;
class CompDemo2
{       public static void main(String args[])
        {       // Pass a reverse comparator to TreeSet() via a lambda expression.
TreeSet<String> ts = new TreeSet<String>((aStr, bStr) -> bStr.compareTo(aStr));
                        ts.add("C");
                        ts.add("A");
                        ts.add("B");
                        ts.add("E");
                        ts.add("F");
                        ts.add("D");
                // Display the elements.
                for(String element : ts)
                        System.out.print(element + " ");
                        System.out.println();
        }
}
```

```
// Use a comparato r to sort accounts by last name.
import java.util.*;
// Compare last whole words in two strings.
class TComp implements Comparator<String>
{       public int compare(String aStr, String bStr)
        {       int i, j, k;
                // Find index of beginning of last name.
                i = aStr.lastIndexOf(' ');
                j = bStr.lastIndexOf(' ');
                k = aStr.substring(i).compareToIgnoreCase (bStr.substring(j));
                if(k==0) // last names match, check entire name
                        return aStr.compareToIgnoreCase (bStr);
                else
                        return k;
        }// No need to override equals.
}
class TreeMapDemo2
{       public static void main(String args[])
        {       // Create a tree map.
TreeMap<String, Double> tm = new TreeMap<String, Double>(new TComp());
                // Put elements to the map.
                        tm.put("John Doe", new Double(3434.34));
                        tm.put("Tom Smith", new Double(123.22));
                        tm.put("Jane Baker", new Double(1378.00));
                        tm.put("Tod Hall", new Double(99.22));
                        tm.put("Ralph Smith", new Double(-19.08));
                // Get a set of the entries.
                        Set<Map.Entry<String, Double>> set = tm.entrySet();
                // Display the elements.
                        for(Map.Entry<String, Double> me : set)
                        {       System.out.print(me.getKey() + ": ");
                                System.out.println(me.getValue());
                        }
                        System.out.println();
        }
}
```

Here is the output; notice that the accounts are now sorted by last name:

```
Jane Baker: 1378.0
John Doe: 3434.34
Todd Hall: 99.22
Ralph Smith: -19.08
Tom Smith: 123.22
```

```
// Use thenComparing() to sort by last, then first name.
import java.util.*;
// A comparator that compares last names.
class CompLastNames implements Comparator<String>
{       public int compare(String aStr, String bStr)
        {       int i, j;
                // Find index of beginning of last name.
                i = aStr.lastIndexOf(' ');
                j = bStr.lastIndexOf(' ');
                return aStr.substring(i).compareToIgnoreCase(bStr.substring(j));
        }
}
// Sort by entire name when last names are equal.
class CompThenByFirstName implements Comparator<String>
{       public int compare(String aStr, String bStr)
        {       int i, j;
                return aStr.compareToIgnoreCase(bStr);
        }
}
class TreeMapDemo2A
{       public static void main(String args[])
        {       // Use thenComparing() to create a comparator that compares
                // last names, then compares entire name when last names match.
                CompLastNames compLN = new CompLastNames();
        Comparator<String> compLastThenFirst =
                compLN.thenComparing(new CompThenByFirstName());
                TreeMap<String, Double> tm =
                        new TreeMap<String, Double>(compLastThenFirst);
                        tm.put("John Doe", new Double(3434.34));
                        tm.put("Tom Smith", new Double(123.22));
                        tm.put("Jane Baker", new Double(1378.00));
                        tm.put("Tod Hall", new Double(99.22));
                        tm.put("Ralph Smith", new Double(-19.08));
                // Get a set of the entries.
                        Set<Map.Entry<String, Double>> set = tm.entrySet();
                for(Map.Entry<String, Double> me : set)
                {       System.out.print(me.getKey() + ": ");
                        System.out.println(me.getValue());
                }
                System.out.println();
        }
}
```

The Collection Algorithms

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

They are summarized in Table.

Method	Description
static <T> boolean addAll(Collection <? super T> *c*, T... *elements*)	Inserts the elements specified by *elements* into the collection specified by *c*. Returns **true** if the elements were added and **false** otherwise.
static <T> Queue<T> asLifoQueue(Deque<T> *c*)	Returns a last-in, first-out view of *c*.
static <T> int binarySearch(List<? extends T> *list*, T *value*, Comparator<? super T> *c*)	Searches for *value* in *list* ordered according to *c*. Returns the position of value in *list*, or a negative value if *value* is not found.
static <T> int binarySearch(List<? extends Comparable<? super T>> *list*, T *value*)	Searches for *value* in *list*. The list must be sorted. Returns the position of *value* in *list*, or a negative value if *value* is not found.
static <E> Collection<E> checkedCollection(Collection<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a collection. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <E> List<E> checkedList(List<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **List**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <K, V> Map<K, V> checkedMap(Map<K, V> *c*, Class<K> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **Map**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <K, V> NavigableMap<K, V> checkedNavigableMap(NavigableMap<K, V> *nm*, Class<E> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **NavigableMap**. An attempt to insert an incompatible element will cause a **ClassCastException**. (Added by JDK 8.)
static <E> NavigableSet<E> checkedNavigableSet(NavigableSet<E> *ns*, Class<E> *t*)	Returns a run-time type-safe view of a **NavigableSet**. An attempt to insert an incompatible element will cause a **ClassCastException**. (Added by JDK 8.)
static <E> Queue<E> checkedQueue(Queue<E> *q*, Class<E> *t*)	Returns a run-time type-safe view of a **Queue**. An attempt to insert an incompatible element will cause a **ClassCastException**. (Added by JDK 8.)
static <E> List<E> checkedSet(Set<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **Set**. An attempt to insert an incompatible element will cause a **ClassCastException**.

Method	Description
static <K, V> SortedMap<K, V> checkedSortedMap(SortedMap<K, V> *c*, Class<K> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **SortedMap**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <E> SortedSet<E> checkedSortedSet(SortedSet<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **SortedSet**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <T> void copy(List<? super T> *list1*, List<? extends T> *list2*)	Copies the elements of *list2* to *list1*.
static boolean disjoint(Collection<?> *a*, Collection<?> *b*)	Compares the elements in *a* to elements in *b*. Returns **true** if the two collections contain no common elements (i.e., the collections contain disjoint sets of elements). Otherwise, returns **false**.
static <T> Enumeration<T> emptyEnumeration( )	Returns an empty enumeration, which is an enumeration with no elements.
static <T> Iterator<T> emptyIterator( )	Returns an empty iterator, which is an iterator with no elements.
static <T> List<T> emptyList( )	Returns an immutable, empty **List** object of the inferred type.
static <T> ListIterator<T> emptyListIterator( )	Returns an empty list iterator, which is a list iterator that has no elements.
static <K, V> Map<K, V> emptyMap( )	Returns an immutable, empty **Map** object of the inferred type.
static <K, V> NavigableMap<K, V> emptyNavigableMap( )	Returns an immutable, empty **NavigableMap** object of the inferred type. (Added by JDK 8.)
static <E> NavigableSet<E> emptyNavigableSet( )	Returns an immutable, empty **NavigableSet** object of the inferred type. (Added by JDK 8.)
static <T> Set<T> emptySet( )	Returns an immutable, empty **Set** object of the inferred type.
static <K, V> SortedMap<K, V> emptySortedMap( )	Returns an immutable, empty **SortedMap** object of the inferred type. (Added by JDK 8.)
static <E> SortedSet<E> emptySortedSet( )	Returns an immutable, empty **SortedSet** object of the inferred type. (Added by JDK 8.)
static <T> Enumeration<T> enumeration(Collection<T> *c*)	Returns an enumeration over *c*. (See "The Enumeration Interface," later in this chapter.)
static <T> void fill(List<? super T> *list*, T *obj*)	Assigns *obj* to each element of *list*.
static int frequency(Collection<?> *c*, object *obj*)	Counts the number of occurrences of *obj* in *c* and returns the result.

Method	**Description**
static int indexOfSubList(List<?> *list*, List<?> *subList*)	Searches *list* for the first occurrence of *subList*. Returns the index of the first match, or –1 if no match is found.
static int lastIndexOfSubList(List<?> *list*, List<?> *subList*)	Searches *list* for the last occurrence of *subList*. Returns the index of the last match, or –1 if no match is found.
static <T> ArrayList<T> list(Enumeration<T> *enum*)	Returns an **ArrayList** that contains the elements of *enum*.
static <T> T max(Collection<? extends T> *c*, Comparator<? super T> *comp*)	Returns the maximum element in *c* as determined by *comp*.
static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> *c*)	Returns the maximum element in *c* as determined by natural ordering. The collection need not be sorted.
static <T> T min(Collection<? extends T> *c*, Comparator<? super T> *comp*)	Returns the minimum element in *c* as determined by *comp*. The collection need not be sorted.
static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> *c*)	Returns the minimum element in *c* as determined by natural ordering.
static <T> List<T> nCopies(int *num*, T *obj*)	Returns *num* copies of *obj* contained in an immutable list. *num* must be greater than or equal to zero.
static <E> Set<E> newSetFromMap(Map<E, Boolean> *m*)	Creates and returns a set backed by the map specified by *m*, which must be empty at the time this method is called.
static <T> boolean replaceAll(List<T> *list*, T *old*, T *new*)	Replaces all occurrences of *old* with *new* in *list*. Returns **true** if at least one replacement occurred. Returns **false** otherwise.
static void reverse(List<T> *list*)	Reverses the sequence in *list*.
static <T> Comparator<T> reverseOrder(Comparator<T> *comp*)	Returns a reverse comparator based on the one passed in *comp*. That is, the returned comparator reverses the outcome of a comparison that uses *comp*.
static <T> Comparator<T> reverseOrder( )	Returns a reverse comparator, which is a comparator that reverses the outcome of a comparison between two elements.
static void rotate(List<T> *list*, int *n*)	Rotates *list* by *n* places to the right. To rotate left, use a negative value for *n*.
static void shuffle(List<T> *list*, Random *r*)	Shuffles (i.e., randomizes) the elements in *list* by using *r* as a source of random numbers.
static void shuffle(List<T> *list*)	Shuffles (i.e., randomizes) the elements in *list*.

Method	Description
static <T> Set<T> singleton(T *obj*)	Returns *obj* as an immutable set. This is an easy way to convert a single object into a set.
static <T> List<T> singletonList(T *obj*)	Returns *obj* as an immutable list. This is an easy way to convert a single object into a list.
static <K, V> Map<K, V> singletonMap(K *k*, V *v*)	Returns the key/value pair *k*/*v* as an immutable map. This is an easy way to convert a single key/value pair into a map.
static <T> void sort(List<T> *list*, Comparator<? super T> *comp*)	Sorts the elements of *list* as determined by *comp*.
static <T extends Comparable<? super T>> void sort(List<T> *list*)	Sorts the elements of *list* as determined by their natural ordering.
static void swap(List<?> *list*, int *idx1*, int *idx2*)	Exchanges the elements in *list* at the indices specified by *idx1* and *idx2*.
static <T> Collection<T> synchronizedCollection(Collection<T> *c*)	Returns a thread-safe collection backed by *c*.
static <T> List<T> synchronizedList(List<T> *list*)	Returns a thread-safe list backed by *list*.
static <K, V> Map<K, V> synchronizedMap(Map<K, V> *m*)	Returns a thread-safe map backed by *m*.
static <K, V> NavigableMap<K, V> synchronizedNavigableMap(NavigableMap<K, V> *nm*)	Returns a synchronized navigable map backed by *nm*. (Added by JDK 8.)
static <T> NavigableSet<T> synchronizedNavigableSet(NavigableSet<T> *ns*)	Returns a synchronized navigable set backed by *ns*. (Added by JDK 8.)
static <T> Set<T> synchronizedSet(Set<T> *s*)	Returns a thread-safe set backed by *s*.
static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> *sm*)	Returns a thread-safe sorted map backed by *sm*.
static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> *ss*)	Returns a thread-safe sorted set backed by *ss*.
static <T> Collection<T> unmodifiableCollection(Collection<? extends T> *c*)	Returns an unmodifiable collection backed by *c*.
static <T> List<T> unmodifiableList(List<? extends T> *list*)	Returns an unmodifiable list backed by *list*.
static <K, V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> *m*)	Returns an unmodifiable map backed by *m*.
static <K, V> NavigableMap<K, V> unmodifiableNavigableMap(NavigableMap<K, ? extends V> *nm*)	Returns an unmodifiable navigable map backed by *nm*. (Added by JDK 8.)
static <T> NavigableSet<T> unmodifiableNavigableSet(NavigableSet<T> *ns*)	Returns an unmodifiable navigable set backed by *ns*. (Added by JDK 8.)
static <T> Set<T> unmodifiableSet(Set<? extends T> *s*)	Returns an unmodifiable set backed by *s*.
static <K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> *sm*)	Returns an unmodifiable sorted map backed by *sm*.
static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> *ss*)	Returns an unmodifiable sorted set backed by *ss*.

Several of the methods can throw a **ClassCastException, UnsupportedOperationException**, Other exceptions are possible, depending on the method.

```
// Demonstrate various algorithms.
import java.util.*;
class AlgorithmsDemo
{       public static void main(String args[])
        {       // Create and initialize linked list.
                        LinkedList<Integer> ll = new LinkedList<Integer>();
                        ll.add(-8);
                        ll.add(20);
                        ll.add(-20);
                        ll.add(8);
                // Create a reverse order comparator.
                        Comparator<Integer> r = Collections.reverseOrder();
                // Sort list by using the comparator.
                        Collections.sort(ll, r);
                System.out.print("List sorted in reverse: ");
                for(int i : ll)
                        System.out.print(i+ " ");
                System.out.println();
                // Shuffle list.
                        Collections.shuffle(ll);
                // Display randomized list.
                        System.out.print("List shuffled: ");
                for(int i : ll)
                        System.out.print(i + " ");
                System.out.println();
                System.out.println("Minimum: " + Collections.min(ll));
                System.out.println("Maximum: " + Collections.max(ll));
        }
}
```

Output from this program is shown here:

```
List sorted in reverse: 20 8 -8 -20
List shuffled: 20 -20 8 -8
Minimum: -20
Maximum: 20
```

Arrays

The **Arrays** class provides various methods that are useful when working with arrays.

- **asList( ):** returns a **List** that is backed by a specified array

  static <T> List asList(T... *array*)

  Here, *array* is the array that contains the data.

- **binarySearch( ):** uses a binary search to find a specified value. This method must be applied to sorted arrays. Here are some of its forms.

  static int binarySearch(byte *array*[ ], byte *value*)
  static int binarySearch(char *array*[ ], char *value*)
  static int binarySearch(double *array*[ ], double *value*)
  static int binarySearch(float *array*[ ], float *value*)
  static int binarySearch(int *array*[ ], int *value*)
  static int binarySearch(long *array*[ ], long *value*)
  static int binarySearch(short *array*[ ], short *value*)
  static int binarySearch(Object *array*[ ], Object *value*)
  static <T> int binarySearch(T[ ] *array*, T *value*, Comparator<? super T> *c*)

  Here, *array* is the array to be searched, and *value* is the value to be located.

  In all cases, if *value* exists in *array*, the index of the element is returned. Otherwise, a negative value is returned

  The last two forms throw a **ClassCastException** if *array* contains elements that cannot be compared (for example, **Double** and **StringBuffer**) or if *value* is not compatible with the types in *array*.

  In the last form, the **Comparator** *c* is used to determine the order of the elements in *array*.

- **copyOf( ):** returns a copy of an array and has the following forms:

  static boolean[ ] copyOf(boolean[ ] *source*, int *len*)
  static byte[ ] copyOf(byte[ ] *source*, int *len*)
  static char[ ] copyOf(char[ ] *source*, int *len*)
  static double[ ] copyOf(double[ ] *source*, int *len*)
  static float[ ] copyOf(float[ ] *source*, int *len*)
  static int[ ] copyOf(int[ ] *source*, int *len*)
  static long[ ] copyOf(long[ ] *source*, int *len*)
  static short[ ] copyOf(short[ ] *source*, int *len*)
  static <T> T[ ] copyOf(T[ ] *source*, int *len*)
  static <T,U> T[ ] copyOf(U[ ] *source*, int *len*, Class<? extends T[ ]> *resultT*)

  The original array is specified by *source*, and the length of the copy is specified by *len*.

If the copy is longer than *source*, then the copy is padded with zeros (for numeric arrays), **null**s (for object arrays), or **false** (for boolean arrays).

If the copy is shorter than *source*, then the copy is truncated.

In the last form, the type of *resultT* becomes the type of the array returned.

If *len* is negative, a **NegativeArraySizeException** is thrown.

If *source* is **null**, a **NullPointerException** is thrown.

If *resultT* is incompatible with the type of *source*, an **ArrayStoreException** is thrown.

- **copyOfRange( ):** returns a copy of a range within an array and has the following forms:

  static boolean[ ] copyOfRange(boolean[ ] *source*, int *start*, int *end*)
  static byte[ ] copyOfRange(byte[ ] *source*, int *start*, int *end*)
  static char[ ] copyOfRange(char[ ] *source*, int *start*, int *end*)
  static double[ ] copyOfRange(double[ ] *source*, int *start*, int *end*)
  static float[ ] copyOfRange(float[ ] *source*, int *start*, int *end*)
  static int[ ] copyOfRange(int[ ] *source*, int *start*, int *end*)
  static long[ ] copyOfRange(long[ ] *source*, int *start*, int *end*)
  static short[ ] copyOfRange(short[ ] *source*, int *start*, int *end*)
  static <T> T[ ] copyOfRange(T[ ] *source*, int *start*, int *end*)
  static <T,U> T[ ] copyOfRange(U[ ] *source*, int *start*, int *end*,
  Class<? extends T[ ]> *resultT*)

  The original array is specified by *source*. The range to copy is specified by the indices passed via *start* and *end*. The range runs from *start* to *end* – 1.

  If the range is longer than *source*, then the copy is padded with zeros (for numeric arrays), **null**s (for object arrays), or **false** (for boolean arrays).

  In the last form, the type of *resultT* becomes the type of the array returned.

  If *start* is negative or greater than the length of *source*, an **IllegalArgumentException** is thrown.

  If *source* is **null**, a **NullPointerException** is thrown. If *resultT* is incompatible with the type of *source*, an **ArrayStoreException** is thrown.

- **equals( ):** returns **true** if two arrays are equivalent. Otherwise, it returns **false**. The **equals( )** method has the following forms:

  static boolean equals(boolean *array1*[ ], boolean *array2* [ ])
  static boolean equals(byte *array1*[ ], byte *array2* [ ])
  static boolean equals(char *array1*[ ], char *array2* [ ])
  static boolean equals(double *array1*[ ], double *array2* [ ])
  static boolean equals(float *array1*[ ], float *array2* [ ])
  static boolean equals(int *array1*[ ], int *array2* [ ])
  static boolean equals(long *array1*[ ], long *array2* [ ])
  static boolean equals(short *array1*[ ], short *array2* [ ])

static boolean equals(Object *array1*[ ], Object *array2* [ ])

Here, *array1* and *array2* are the two arrays that are compared for equality.

- **deepEquals( ):** can be used to determine if two arrays, which might contain nested arrays, are equal. It has this declaration:

  static boolean deepEquals(Object[ ] *a*, Object[ ] *b*)

  It returns **true** if the arrays passed in *a* and *b* contain the same elements. If *a* and *b* contain nested arrays, then the contents of those nested arrays are also checked.

  It returns **false** if the arrays, or any nested arrays, differ.

- **fill( ):** assigns a value to all elements in an array. In other words, it fills an array with a specified value. The **fill( )** method has two versions.
  - The first version, which has the following forms, fills an entire array:

    static void fill(boolean *array*[ ], boolean *value*)
    static void fill(byte *array*[ ], byte *value*)
    static void fill(char *array*[ ], char *value*)
    static void fill(double *array*[ ], double *value*)
    static void fill(float *array*[ ], float *value*)
    static void fill(int *array*[ ], int *value*)
    static void fill(long *array*[ ], long *value*)
    static void fill(short *array*[ ], short *value*)
    static void fill(Object *array*[ ], Object *value*)

    Here, *value* is assigned to all elements in *array*.
  - The second version of the **fill( )** method assigns a value to a subset of an array.

- **sort( ):** sorts an array so that it is arranged in ascending order. The **sort( )** method has two versions.
  - The first version, shown here, sorts the entire array:

    static void sort(byte *array*[ ])
    static void sort(char *array*[ ])
    static void sort(double *array*[ ])
    static void sort(float *array*[ ])
    static void sort(int *array*[ ])
    static void sort(long *array*[ ])
    static void sort(short *array*[ ])
    static void sort(Object *array*[ ])
    static <T> void sort(T *array*[ ], Comparator<? super T> *c*)

    Here, *array* is the array to be sorted.

In the last form, *c* is a **Comparator** that is used to order the elements of *array*.

The last two forms can throw a **ClassCastException** if elements of the array being sorted are not comparable.

- The second version of **sort( )** enables you to specify a range within an array that you want to sort.

JDK 8 adds several new methods to **Arrays**.

- **parallelSort( ):** because it sorts, into ascending order, portions of an array in parallel and then merges the results. This approach can greatly speed up sorting times. There are two basic types each with several overloads.
    - The first type sorts the entire array. It is shown here:

      static void parallelSort(byte *array*[ ])
      static void parallelSort(char *array*[ ])
      static void parallelSort(double *array*[ ])
      static void parallelSort(float *array*[ ])
      static void parallelSort(int *array*[ ])
      static void parallelSort(long *array*[ ])
      static void parallelSort(short *array*[ ])
      static <T extends Comparable<? super T>> void parallelSort(T *array*[ ])
      static <T> void parallelSort(T *array*[ ], Comparator<? super T> *c*)

      Here, *array* is the array to be sorted.
      In the last form, *c* is a comparator that is used to order the elements in the array.
      The last two forms can throw a **ClassCastException** if the elements of the array being sorted are not comparable.
    - The second version of **parallelSort( )** enables you to specify a range within the array that you want to sort.

Beginning with JDK 8, **Arrays** supports the new **Stream** interface by

- **stream( ):** It has two forms.
    - The first is shown here:

      static DoubleStream stream(double *array*[ ])
      static IntStream stream(int *array*[ ])
      static LongStream stream(long *array*[ ])
      static <T> Stream stream(T *array*[ ])

      Here, *array* is the array to which the stream will refer.
    - The second version of **stream( )** enables you to specify a range within an array.

JDK8 adds three more new methods. Two are related: **setAll( )** and **parallelSetAll()** . Both assign values to all of the elements, but **parallelSetAll( )** works in parallel. Here is an example of each:

static void setAll(double *array*[ ],
IntToDoubleFunction<? extends T> *genVal*)
static void parallelSetAll(double *array*[ ],
IntToDoubleFunction<? extends T> *genVal*)

Several overloads exist for each of these that handle types **int**, **long**, and generic.

Finally, JDK 8 includes one of the more intriguing additions to **Arrays**.

- **parallelPrefix( )**: it modifies an array so that each element contains the cumulative result of an operation applied to all previous elements. For example, if the operation is multiplication, then on return, the array elements will contain the values associated with the running product of the original values. It has several overloads. Here is one example:
  static void parallelPrefix(double *array*[ ], DoubleBinaryOperator *func*)

Here, *array* is the array being acted upon, and *func* specifies the operation applied.

Arrays also provides **toString( )** and **hashCode( )** for the various types of arrays. In addition, **deepToString( )** and **deepHashCode( )** are provided, which operate effectively on arrays that contain nested arrays.

```
// Demonstrate Arrays
import java.util.*;
class ArraysDemo
{       public static void main(String args[])
        {       // Allocate and initialize array.
                        int array[] = new int[10];
                        for(int i = 0; i < 10; i++)
                                array[i] = -3 * i;
                // Display, sort, and display the array.
                        System.out.print("Original contents: ");
                        display(array);
                        Arrays.sort(array);
                        System.out.print("Sorted: ");
                        display(array);
                // Fill and display the array.
                        Arrays.fill(array, 2, 6, -1);
                        System.out.print("After fill(): ");
                        display(array);
                // Sort and display the array.
                        Arrays.sort(array);
                        System.out.print("After sorting again: ");
                        display(array);
                // Binary search for -9.
                        System.out.print("The value -9 is at location ");
                        int index =Arrays.binarySearch(array, -9);
                        System.out.println(index);
        }
        static void display(int array[])
        {       for(int i: array)
                        System.out.print(i + " ");
                System.out.println();
        }
}
```

```
Original contents: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Sorted: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
After fill(): -27 -24 -1 -1 -1 -1 -9 -6 -3 0
After sorting again: -27 -24 -9 -6 -3 -1 -1 -1 -1 0
The value -9 is at location 2
```

The Legacy Classes and Interfaces

Early versions of **java.util** did not include the Collections Framework. Instead, it defined several classes and an interface that provided an ad hoc method of storing objects. When collections were added (by J2SE 1.2), several of the original classes were reengineered to support the collection interfaces. Thus, they are now technically part of the Collections Framework

Vector

- **Vector** implements a dynamic array.
- It is similar to **ArrayList**, but with two differences:
  - **Vector** is synchronized,
  - It contains many legacy methods that duplicate the functionality of methods defined by the Collections Framework.
    - **Vector** is declared like this:

      class Vector<E>

Here, **E** specifies the type of element that will be stored.

Here are the **Vector** constructors:

Vector( ): creates a default vector, which has an initial size of 10

Vector(int *size*): creates a vector whose initial capacity is specified by *size*.

Vector(int *size*, int *incr*): creates a vector whose initial capacity is specified by *size* and whose increment is specified by *incr*. The increment specifies the number of elements to allocate each time that a vector is resized upward.

Vector(Collection<? extends E> *c*): creates a vector that contains the elements of collection *c*.

All vectors start with an initial capacity. After this initial capacity is reached, the next time that you attempt to store an object in the vector, the vector automatically allocates space for that object plus extra room for additional objects. By allocating more than just the required memory, the vector reduces the number of allocations that must take place as the vector grows. This reduction is important, because allocations are costly in terms of time. The amount of extra space allocated during each reallocation is determined by the increment that you specify when you create the vector. If you don't specify an increment, the vector's size is doubled by each allocation cycle.

Vector defines these protected data members:

int capacityIncrement; The increment value is stored in **capacityIncrement**

int elementCount; The number of elements currently in the vector is stored in **elementCount**

Object[ ] elementData; The array that holds the vector is stored in **elementData**.

Vector defines several legacy methods, which are summarized in Table

Method	Description
void addElement(E *element*)	The object specified by *element* is added to the vector.
int capacity( )	Returns the capacity of the vector.
Object clone( )	Returns a duplicate of the invoking vector.
boolean contains(Object *element*)	Returns **true** if *element* is contained by the vector, and returns **false** if it is not.
void copyInto(Object *array*[ ])	The elements contained in the invoking vector are copied into the array specified by *array*.
E elementAt(int *index*)	Returns the element at the location specified by *index*.
Enumeration<E> elements( )	Returns an enumeration of the elements in the vector.
void ensureCapacity(int *size*)	Sets the minimum capacity of the vector to *size*.
E firstElement( )	Returns the first element in the vector.
int indexOf(Object *element*)	Returns the index of the first occurrence of *element*. If the object is not in the vector, –1 is returned.
int indexOf(Object *element*, int *start*)	Returns the index of the first occurrence of *element* at or after *start*. If the object is not in that portion of the vector, –1 is returned.
void insertElementAt(E *element*, int *index*)	Adds *element* to the vector at the location specified by *index*.
boolean isEmpty( )	Returns **true** if the vector is empty, and returns **false** if it contains one or more elements.
E lastElement( )	Returns the last element in the vector.
int lastIndexOf(Object *element*)	Returns the index of the last occurrence of *element*. If the object is not in the vector, –1 is returned.
int lastIndexOf(Object *element*, int *start*)	Returns the index of the last occurrence of *element* before *start*. If the object is not in that portion of the vector, –1 is returned.
void removeAllElements( )	Empties the vector. After this method executes, the size of the vector is zero.
boolean removeElement(Object *element*)	Removes *element* from the vector. If more than one instance of the specified object exists in the vector, then it is the first one that is removed. Returns **true** if successful and **false** if the object is not found.
void removeElementAt(int *index*)	Removes the element at the location specified by *index*.
void setElementAt(E *element*, int *index*)	The location specified by *index* is assigned *element*.
void setSize(int *size*)	Sets the number of elements in the vector to *size*. If the new size is less than the old size, elements are lost. If the new size is larger than the old size, **null** elements are added.
int size( )	Returns the number of elements currently in the vector.
String toString( )	Returns the string equivalent of the vector.
void trimToSize( )	Sets the vector's capacity equal to the number of elements that it currently holds.

```
// Demonstrate various Vector operations.
import java.util.*;
class VectorDemo
{       public static void main(String args[])
        {       // initial size is 3, increment is 2
                Vector<Integer> v = new Vector<Integer>(3, 2);
                System.out.println("Initial size: " + v.size());
                System.out.println("Initial capacity: " +v.capacity());
                v.addElement(1);
                v.addElement(2);
                v.addElement(3);
                v.addElement(4);
                System.out.println("Capacity after four additions: " +v.capacity());
                v.addElement(5);
                System.out.println("Current capacity: " +v.capacity());
                v.addElement(6);
                v.addElement(7);
                System.out.println("Current capacity: " +v.capacity());
                v.addElement(9);
                v.addElement(10);
                System.out.println("Current capacity: " +v.capacity());
                v.addElement(11);
                v.addElement(12);
                System.out.println("First element: " + v.firstElement());
                System.out.println("Last element: " + v.lastElement());
                if(v.contains(3))
                        System.out.println("Vector contains 3.");
                // Enumerate the elements in the vector.
                        Enumeration<Integer> vEnum = v.elements();
                System.out.println("\nElements in vector:");
                while(vEnum.hasMoreElements())
                        System.out.print(vEnum.nextElement() + " ");
                System.out.println();
        }
}
```
The output from this program is shown here:
```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
```

```
First element: 1
Last element: 12
Vector contains 3.
Elements in vector:
1 2 3 4 5 6 7 9 10 11 12
```

the following iterator-based code can be substituted into the program:

```
// Use an iterator to display contents.
        Iterator<Integer> vItr = v.iterator();
        System.out.println("\nElements in vector:");
        while(vItr.hasNext())
        System.out.print(vItr.next() + " ");
        System.out.println();
```

You can also use a for-each **for** loop to cycle through a **Vector**, as the following version of the preceding code shows:

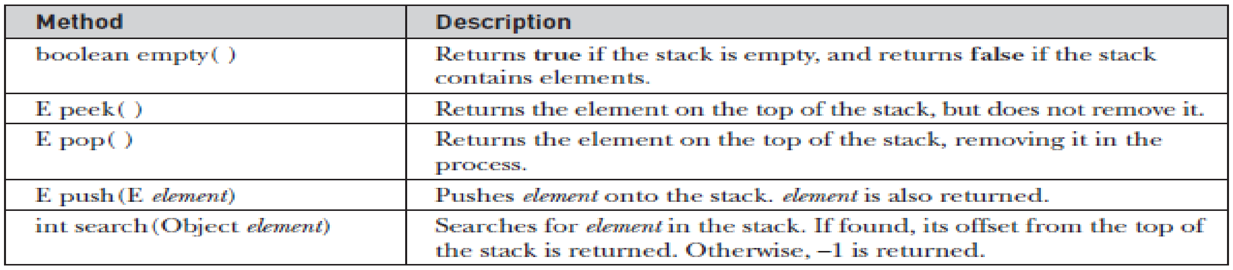
```
// Use an enhanced for loop to display contents
        System.out.println("\nElements in vector:");
        for(int i : v)
                System.out.print(i + " ");
        System.out.println();
```

Stack

- **Stack** is a subclass of **Vector** that implements a standard last-in, first-out stack.
- **Stack** only defines the default constructor, which creates an empty stack.
- With the release of JDK 5, **Stack** was retrofitted for generics and is declared as shown here: class Stack<E>

  Here, **E** specifies the type of element stored in the stack.

Stack includes all the methods defined by **Vector** and adds several of its own, shown in Table



Method	Description
boolean empty( )	Returns **true** if the stack is empty, and returns **false** if the stack contains elements.
E peek( )	Returns the element on the top of the stack, but does not remove it.
E pop( )	Returns the element on the top of the stack, removing it in the process.
E push(E *element*)	Pushes *element* onto the stack. *element* is also returned.
int search(Object *element*)	Searches for *element* in the stack. If found, its offset from the top of the stack is returned. Otherwise, –1 is returned.

```
// Demonstrate the Stack class.
import java.util.*;
class StackDemo
```

```
{       static void showpush(Stack<Integer> st, int a)
        {       st.push(a);
                System.out.println("push(" + a + ")");
                System.out.println("stack: " + st);
        }
        static void showpop(Stack<Integer> st)
        {       System.out.print("pop -> ");
                Integer a = st.pop();
                System.out.println(a);
                System.out.println("stack: " + st);
        }
        public static void main(String args[])
        {       Stack<Integer> st = new Stack<Integer>();
                System.out.println("stack: " + st);
                showpush(st, 42);
                showpush(st, 66);
                showpush(st, 99);
                showpop(st);
                showpop(st);
                showpop(st);
                try
                {       showpop(st);
                }
                catch (EmptyStackException e)
                {       System.out.println("empty stack");
                }
        }
}
```

```
Output:         stack: [ ]
                push(42)
                stack: [42]
                push(66)
                stack: [42, 66]
                push(99)
                stack: [42, 66, 99]
                pop -> 99
                stack: [42, 66]
                pop -> 66
                stack: [42]
                pop -> 42
                stack: [ ]
                pop -> empty stack
```

Dictionary

- **Dictionary** is an abstract class that represents a key/value storage repository and operates much like **Map**.
- With the advent of JDK 5, **Dictionary** was made generic. It is declared as shown here: class Dictionary<K, V>
  Here, **K** specifies the type of keys, and **V** specifies the type of values

The abstract methods defined by **Dictionary** are listed in Table.

Method	Purpose
Enumeration<V> elements( )	Returns an enumeration of the values contained in the dictionary.
V get(Object *key*)	Returns the object that contains the value associated with *key*. If *key* is not in the dictionary, a **null** object is returned.
boolean isEmpty( )	Returns **true** if the dictionary is empty, and returns **false** if it contains at least one key.
Enumeration<K> keys( )	Returns an enumeration of the keys contained in the dictionary.
V put(K *key*, V *value*)	Inserts a key and its value into the dictionary. Returns **null** if *key* is not already in the dictionary; returns the previous value associated with *key* if *key* is already in the dictionary.
V remove(Object *key*)	Removes *key* and its value. Returns the value associated with *key*. If *key* is not in the dictionary, a **null** is returned.
int size( )	Returns the number of entries in the dictionary.

Hashtable

- It is similar to **HashMap**, but is synchronized.
- **Hashtable** does not directly support iterators.
- Like **HashMap**, **Hashtable** stores key/value pairs in a hash table.
- A hash table can only store objects that override the **hashCode( )** and **equals( )** methods that are defined by **Object**.
- The **hashCode( )** must compute and return the hash code for the object.
- **equals( )** compares two objects.
- **Hashtable** was made generic by JDK 5. It is declared like this:
  class Hashtable<K, V>
  Here, **K** specifies the type of keys, and **V** specifies the type of values.

The **Hashtable** constructors are shown here:
Hashtable( ): the default constructor
Hashtable(int *size*): creates a hash table that has an initial size specified by *size*. (The default size is 11.)

Hashtable(int *size*, float *fillRatio*): creates a hash table that has an initial size specified by *size* and a fill ratio specified by *fillRatio*. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash table multiplied by its fill ratio, the hash table is expanded. If you do not specify a fill ratio, then 0.75 is used.

Hashtable(Map<? extends K, ? extends V> *m*): creates a hash table that is initialized with the elements in *m*. The default load factor of 0.75 is used.

Hashtable defines the legacy methods listed in Table.

Method	Description
void clear( )	Resets and empties the hash table.
Object clone( )	Returns a duplicate of the invoking object.
boolean contains(Object *value*)	Returns **true** if some value equal to *value* exists within the hash table. Returns **false** if the value isn't found.
boolean containsKey(Object *key*)	Returns **true** if some key equal to *key* exists within the hash table. Returns **false** if the key isn't found.
boolean containsValue(Object *value*)	Returns **true** if some value equal to *value* exists within the hash table. Returns **false** if the value isn't found.
Enumeration<V> elements( )	Returns an enumeration of the values contained in the hash table.
V get(Object *key*)	Returns the object that contains the value associated with *key*. If *key* is not in the hash table, a **null** object is returned.
boolean isEmpty( )	Returns **true** if the hash table is empty; returns **false** if it contains at least one key.
Enumeration<K> keys( )	Returns an enumeration of the keys contained in the hash table.
V put(K *key*, V *value*)	Inserts a key and a value into the hash table. Returns **null** if *key* isn't already in the hash table; returns the previous value associated with *key* if *key* is already in the hash table.
void rehash( )	Increases the size of the hash table and rehashes all of its keys.
V remove(Object *key*)	Removes *key* and its value. Returns the value associated with *key*. If *key* is not in the hash table, a **null** object is returned.
int size( )	Returns the number of entries in the hash table.
String toString( )	Returns the string equivalent of a hash table.

```
// Demonstrate a Hashtable.
import java.util.*;
class HTDemo
{       public static void main(String args[])
        {  Hashtable<String, Double> balance = new Hashtable<String, Double>();
```

```
                Enumeration<String> names;
                String str;
                double bal;
                balance.put("John Doe", 3434.34);
                balance.put("Tom Smith", 123.22);
                balance.put("Jane Baker", 1378.00);
                balance.put("Tod Hall", 99.22);
                balance.put("Ralph Smith", -19.08);
                // Show all balances in hashtable.
                        names = balance.keys();
                        while(names.hasMoreElements())
                        {       str = names.nextElement();
                                System.out.println(str + ": " +balance.get(str));
                        }
                        System.out.println();
                }
}
```

The output from this program is shown here:

```
Todd Hall: 99.22
Ralph Smith: -19.08
John Doe: 3434.34
Jane Baker: 1378.0
Tom Smith: 123.22
```

One important point: Like the map classes, **Hashtable** does not directly support iterators. Thus, the preceding program uses an enumeration to display the contents of balance. However, you can obtain set-views of the hash table, which permits the use of iterators. To do so, you simply use one of the collection-view methods defined by **Map**, such as **entrySet( )** or **keySet( )**. For example, you can obtain a set-view of the keys and cycle through them using either an iterator or an enhanced **for** loop. Here is a reworked version of the program that shows this technique:

```
// Use iterators with a Hashtable.
import java.util.*;
class HTDemo2
{       public static void main(String args[])
        {  Hashtable<String, Double> balance =new Hashtable<String, Double>();
                String str;
                double bal;
                balance.put("John Doe", 3434.34);
                balance.put("Tom Smith", 123.22);
                balance.put("Jane Baker", 1378.00);
                balance.put("Tod Hall", 99.22);
                balance.put("Ralph Smith", -19.08);
                // Show all balances in hashtable.
                // First, get a set view of the keys.
                        Set<String> set = balance.keySet();
                // Get an iterator.
                        Iterator<String> itr = set.iterator();
                        while(itr.hasNext())
                        {       str = itr.next();
                                System.out.println(str + ": " +balance.get(str));
                        }
                        System.out.println();
                }
}
```

Properties

- **Properties** is a subclass of **Hashtable**.
- It is used to maintain lists of values in which the key is a **String** and the value is also a **String**.
- The **Properties** class is used by some other Java classes. For example, it is the type of object returned by **System.getProperties( )** when obtaining environmental values.
- Although the **Properties** class, itself, is not generic, several of its methods are.

Properties defines the following instance variable:

Properties defaults;	This variable holds a default property list associated with a **Properties** object.

Properties defines these constructors:

Properties( ): creates a **Properties** object that has no default values.

Properties(Properties *propDefault*): creates an object that uses *propDefault* for its default values

In both cases, the property list is empty.

In addition to the methods that **Properties** inherits from **Hashtable**, **Properties** defines the methods listed in Table

Method	Description
String getProperty(String *key*)	Returns the value associated with *key*. A **null** object is returned if *key* is neither in the list nor in the default property list.
String getProperty(String *key*, String *defaultProperty*)	Returns the value associated with *key*. *defaultProperty* is returned if *key* is neither in the list nor in the default property list.
void list(PrintStream *streamOut*)	Sends the property list to the output stream linked to *streamOut*.
void list(PrintWriter *streamOut*)	Sends the property list to the output stream linked to *streamOut*.
void load(InputStream *streamIn*) throws IOException	Inputs a property list from the input stream linked to *streamIn*.
void load(Reader *streamIn*) throws IOException	Inputs a property list from the input stream linked to *streamIn*.
void loadFromXML(InputStream *streamIn*) throws IOException, InvalidPropertiesFormatException	Inputs a property list from an XML document linked to *streamIn*.
Enumeration<?> propertyNames( )	Returns an enumeration of the keys. This includes those keys found in the default property list, too.
Object setProperty(String *key*, String *value*)	Associates *value* with *key*. Returns the previous value associated with *key*, or returns **null** if no such association exists.
void store(OutputStream *streamOut*, String *description*) throws IOException	After writing the string specified by *description*, the property list is written to the output stream linked to *streamOut*.
void store(Writer *streamOut*, String *description*) throws IOException	After writing the string specified by *description*, the property list is written to the output stream linked to *streamOut*.
void storeToXML(OutputStream *streamOut*, String *description*) throws IOException	After writing the string specified by *description*, the property list is written to the XML document linked to *streamOut*.
void storeToXML(OutputStream *streamOut*, String *description*, String *enc*)	The property list and the string specified by *description* is written to the XML document linked to *streamOut* using the specified character encoding.
Set<String> stringPropertyNames( )	Returns a set of keys.

Properties also contains one deprecated method: **save( )**. This was replaced by **store( )** because **save( )** did not handle errors correctly.

One useful capability of the **Properties** class is that you can specify a default property that will be returned if no value is associated with a certain key. For example, a default value can be specified along with the key in the **getProperty( )** method—such as **getProperty("name" ,"default value")**. If the "name" value is not found, then "default value" is returned. When you construct a **Properties** object, you can pass another instance of **Properties** to be used as the default properties for the new instance. In this case, if you call **getProperty("foo")** on a given **Properties** object, and "foo" does not exist, Java looks for "foo" in the default **Properties** object. This allows for arbitrary nesting of levels of default properties.

// Demonstrate a Property list.

```
import java.util.*;
class PropDemo
{       public static void main(String args[])
        {       Properties capitals = new Properties();
                capitals.put("Illinois", "Springfield");
                capitals.put("Missouri", "Jefferson City");
                capitals.put("Washington", "Olympia");
                capitals.put("California", "Sacramento");
                capitals.put("Indiana", "Indianapolis");
                // Get a set-view of the keys.
                        Set<?> states = capitals.keySet();
                // Show all of the states and capitals.
                        for(Object name : states)
                                System.out.println("The capital of " +name + " is " +
                                              capitals.getProperty((String)name)+ ".");
                                System.out.println();
                // Look for state not in list -- specify default.
                        String str = capitals.getProperty("Florida", "Not Found");
                        System.out.println("The capital of Florida is " + str + ".");
        }
}
```

The output from this program is shown here:

```
The capital of Missouri is Jefferson City.
The capital of Illinois is Springfield.
The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.
The capital of Florida is Not Found.
```

Since Florida is not in the list, the default value is used.

```
// Use a default property list.
import java.util.*;
class PropDemoDef
{       public static void main(String args[])
        {       Properties defList = new Properties();
                defList.put("Florida", "Tallahassee");
                defList.put("Wisconsin", "Madison");
                Properties capitals = new Properties(defList);
                capitals.put("Illinois", "Springfield");
                capitals.put("Missouri", "Jefferson City");
                capitals.put("Washington", "Olympia");
                capitals.put("California", "Sacramento");
                capitals.put("Indiana", "Indianapolis");
                // Get a set-view of the keys.
                        Set<?> states = capitals.keySet();
                // Show all of the states and capitals.
                        for(Object name : states)
                                System.out.println("The capital of " +name + " is " +
                                        capitals.getProperty((String)name)+ ".");
                        System.out.println();
                // Florida will now be found in the default list.
                        String str = capitals.getProperty("Florida");
                System.out.println("The capital of Florida is "+ str + ".");
        }
}
```

Output:

```
The capital of Missouri is Jefferson City.
The capital of Illinois is Springfield.
The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.
The capital of Florida is Tallahassee.
```

Using store( ) and load( )

One of the most useful aspects of **Properties** is that the information contained in a **Properties** object can be easily stored to or loaded from disk with the **store( )** and **load( )** methods. At any time, you can write a **Properties** object to a stream or read it back. This makes property lists especially convenient for implementing simple databases.

```
/* A simple telephone number database that uses a property list. */
import java.io.*;
import java.util.*;
class Phonebook
{       public static void main(String args[])  throws IOException
        {       Properties ht = new Properties();
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                String name, number;
                FileInputStream fin = null;
                boolean changed = false;
                // Try to open phonebook.dat file.
                try
                {       fin = new FileInputStream("phonebook.dat");
                }
```

```
		catch(FileNotFoundException e)
		{	// ignore missing file
		}
		/* If phonebook file already exists,load existing telephone numbers. */
		try
		{	if(fin != null)
			{	ht.load(fin);
				fin.close();
			}
		}
		catch(IOException e)
		{	System.out.println("Error reading file.");
		}
		// Let user enter new names and numbers.
		do
		{	System.out.println("Enter new name" +" ('quit' to stop): ");
			name = br.readLine();
			if(name.equals("quit")) continue;
				System.out.println("Enter number: ");
			number = br.readLine();
			ht.put(name, number);
			changed = true;
		}
		while(!name.equals("quit"));
		// If phone book data has changed, save it.
		if(changed)
		{
		FileOutputStream fout = new FileOutputStream("phonebook.dat");
			ht.store(fout, "Telephone Book");
			fout.close();
		}
		// Look up numbers given a name.
		do
		{	System.out.println("Enter name to find" +" ('quit' to quit): ");
			name = br.readLine();
			if(name.equals("quit")) continue;
			number = (String) ht.get(name);
			System.out.println(number);
		}
		while(!name.equals("quit"));
	}
}
```

Output:

```
Enter new name ('quit' to stop):
fdfdfd
Enter number:
45434
Enter new name ('quit' to stop):
dfdfd
Enter number:
6454
Enter new name ('quit' to stop):
quit
Enter name to find ('quit' to quit):
fdfdfd
45434
Enter name to find ('quit' to quit):
quit
```

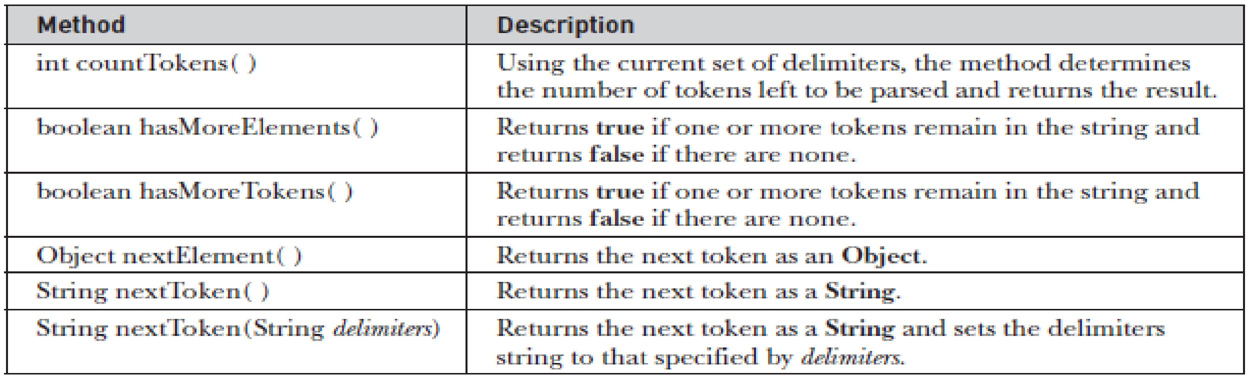
More Utility Classes

StringTokenizer

- The processing of text often consists of parsing a formatted input string. *Parsing* is the division of text into a set of discrete parts, or *tokens*, which in a certain sequence can convey a semantic meaning.
- The **StringTokenizer** class provides the first step in this parsing process, often called the *lexer* (lexical analyzer) or *scanner*.
- **StringTokenizer** implements the **Enumeration** interface.
- To use **StringTokenizer**, you specify an input string and a string that contains delimiters. *Delimiters* are characters that separate tokens. Each character in the delimiters string is considered a valid delimiter—for example, **",;:"** sets the delimiters to a comma, semicolon, and colon. The default set of delimiters consists of the whitespace characters: space, tab, form feed, newline, and carriage return.

The **StringTokenizer** constructors are shown here:
StringTokenizer(String *str*): the default delimiters are used.
StringTokenizer(String *str*, String *delimiters*): *delimiters* is a string that specifies the delimiters.
StringTokenizer(String *str*, String *delimiters*, boolean *delimAsToken*): if *delimAsToken* is **true**, then the delimiters are also returned as tokens when the string is parsed. Otherwise, the delimiters are not returned.Delimiters are not returned as tokens by the first two forms.
In all versions, *str* is the string that will be tokenized.

Once you have created a **StringTokenizer** object, the **nextToken( )** method is used to extract consecutive tokens. The **hasMoreTokens( )** method returns **true** while there are more tokens to be extracted. Since **StringTokenizer** implements **Enumeration**, the **hasMoreElements( )** and **nextElement( )** methods are also implemented, and they act the same as **hasMoreTokens( )** and **nextToken( )**, respectively. The **StringTokenizer** methods are shown in Table.



Method	Description
int countTokens( )	Using the current set of delimiters, the method determines the number of tokens left to be parsed and returns the result.
boolean hasMoreElements( )	Returns **true** if one or more tokens remain in the string and returns **false** if there are none.
boolean hasMoreTokens( )	Returns **true** if one or more tokens remain in the string and returns **false** if there are none.
Object nextElement( )	Returns the next token as an **Object**.
String nextToken( )	Returns the next token as a **String**.
String nextToken(String *delimiters*)	Returns the next token as a **String** and sets the delimiters string to that specified by *delimiters*.

Here is an example that creates a **StringTokenizer** to parse "key=value" pairs.

Consecutive sets of "key=value" pairs are separated by a semicolon.

```
// Demonstrate StringTokenizer.
import java.util.StringTokenizer;
class STDemo
{	static String in = "title=Java: The Complete Reference;" +"author=Schildt;"
	+"publisher=McGraw-Hill;" +"copyright=2014";
	public static void main(String args[])
	{	StringTokenizer st = new StringTokenizer(in, "=;");
		while(st.hasMoreTokens())
		{	String key = st.nextToken();
			String val = st.nextToken();
			System.out.println(key + "\t" + val);
		}
	}
}
```

The output from this program is shown here:

```
title Java: The Complete Reference
author Schildt
publisher McGraw-Hill
copyright 2014
```

BitSet

- A **BitSet** class creates a special type of array that holds bit values in the form of **boolean** values.
- This array can increase in size as needed. This makes it similar to a vector of bits.

The **BitSet** constructors are shown here:
BitSet( ): creates a default object
BitSet(int *size*): specify its initial size (that is, the number of bits that it can hold). All bits are initialized to **false**.

BitSet defines the methods listed in Table below

Method	Description
void and(BitSet *bitSet*)	ANDs the contents of the invoking **BitSet** object with those specified by *bitSet*. The result is placed into the invoking object.
void andNot(BitSet *bitSet*)	For each set bit in *bitSet*, the corresponding bit in the invoking **BitSet** is cleared.
int cardinality( )	Returns the number of set bits in the invoking object.
void clear( )	Zeros all bits.
void clear(int *index*)	Zeros the bit specified by *index*.
void clear(int *startIndex*, int *endIndex*)	Zeros the bits from *startIndex* to *endIndex*–1.
Object clone( )	Duplicates the invoking **BitSet** object.
boolean equals(Object *bitSet*)	Returns **true** if the invoking bit set is equivalent to the one passed in *bitSet*. Otherwise, the method returns **false**.
void flip(int *index*)	Reverses the bit specified by *index*.
void flip(int *startIndex*, int *endIndex*)	Reverses the bits from *startIndex* to *endIndex*–1.
boolean get(int *index*)	Returns the current state of the bit at the specified index.
BitSet get(int *startIndex*, int *endIndex*)	Returns a **BitSet** that consists of the bits from *startIndex* to *endIndex*–1. The invoking object is not changed.
int hashCode( )	Returns the hash code for the invoking object.
boolean intersects(BitSet *bitSet*)	Returns **true** if at least one pair of corresponding bits within the invoking object and *bitSet* are set.
boolean isEmpty( )	Returns **true** if all bits in the invoking object are cleared.
int length( )	Returns the number of bits required to hold the contents of the invoking **BitSet**. This value is determined by the location of the last set bit.
int nextClearBit(int *startIndex*)	Returns the index of the next cleared bit (that is, the next **false** bit), starting from the index specified by *startIndex*.
int nextSetBit(int *startIndex*)	Returns the index of the next set bit (that is, the next **true** bit), starting from the index specified by *startIndex*. If no bit is set, –1 is returned.
void or(BitSet *bitSet*)	ORs the contents of the invoking **BitSet** object with that specified by *bitSet*. The result is placed into the invoking object.
int previousClearBit(int *startIndex*)	Returns the index of the next cleared bit (that is, the next **false** bit) at or prior to the index specified by *startIndex*. If no cleared bit is found, –1 is returned.
int previousSetBit(int *startIndex*)	Returns the index of the next set bit (that is, the next **true** bit) at or prior to the index specified by *startIndex*. If no set bit is found, –1 is returned.
void set(int *index*)	Sets the bit specified by *index*.
void set(int *index*, boolean *v*)	Sets the bit specified by *index* to the value passed in *v*. **true** sets the bit; **false** clears the bit.
void set(int *startIndex*, int *endIndex*)	Sets the bits from *startIndex* to *endIndex*–1.
void set(int *startIndex*, int *endIndex*, boolean *v*)	Sets the bits from *startIndex* to *endIndex*–1 to the value passed in *v*. **true** sets the bits; **false** clears the bits.
int size( )	Returns the number of bits in the invoking **BitSet** object.
IntStream stream( )	Returns a stream that contains the bit positions, from low to high, that have set bits. (Added by JDK 8.)
byte[ ] toByteArray( )	Returns a **byte** array that contains the invoking **BitSet** object.
long[ ] toLongArray( )	Returns a **long** array that contains the invoking **BitSet** object.
String toString( )	Returns the string equivalent of the invoking **BitSet** object.
static BitSet valueOf(byte[ ] *v*)	Returns a **BitSet** that contains the bits in *v*.
static BitSet valueOf(ByteBuffer *v*)	Returns a **BitSet** that contains the bits in *v*.
static BitSet valueOf(long[ ] *v*)	Returns a **BitSet** that contains the bits in *v*.
static BitSet valueOf(LongBuffer *v*)	Returns a **BitSet** that contains the bits in *v*.
void xor(BitSet *bitSet*)	XORs the contents of the invoking **BitSet** object with that specified by *bitSet*. The result is placed into the invoking object.

```
// BitSet Demonstration.
import java.util.BitSet;
class BitSetDemo
{	public static void main(String args[])
	{	BitSet bits1 = new BitSet(16);
		BitSet bits2 = new BitSet(16);
		// set some bits
		for(int i=0; i<16; i++)
		{	if((i%2) == 0)
				bits1.set(i);
			if((i%5) != 0)
				bits2.set(i);
		}
		System.out.println("Initial pattern in bits1: ");
		System.out.println(bits1);
		System.out.println("\nInitial pattern in bits2: ");
		System.out.println(bits2);
		// AND bits
			bits2.and(bits1);
			System.out.println("\nbits2 AND bits1: ");
			System.out.println(bits2);
		// OR bits
			bits2.or(bits1);
			System.out.println("\nbits2 OR bits1: ");
			System.out.println(bits2);
		// XOR bits
			bits2.xor(bits1);
			System.out.println("\nbits2 XOR bits1: ");
			System.out.println(bits2);
	}
}
```

When **toString( )** converts a **BitSet** object to its string equivalent, each set bit is represented by its bit position. Cleared bits are not shown.

```
Initial pattern in bits1:
{0, 2, 4, 6, 8, 10, 12, 14}
Initial pattern in bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}
bits2 AND bits1:
{2, 4, 6, 8, 12, 14}
bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}
bits2 XOR bits1:    {    }
```

Date

- The **Date** class encapsulates the current date and time.
- **Date** also implements the **Comparable** interface
- When Java 1.1 was released, many of the functions carried out by the original **Date** class were moved into the **Calendar** and **DateFormat** classes, and as a result, many of the original 1.0 **Date** methods were deprecated. Since the deprecated 1.0 methods should not be used for new code, they are not described here.

Date supports the following non-deprecated constructors:
Date( ): initializes the object with the current date and time
Date(long *millisec*): accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970.

The nondeprecated methods defined by **Date** are shown in Table.

Method	Description
boolean after(Date *date*)	Returns **true** if the invoking **Date** object contains a date that is later than the one specified by *date*. Otherwise, it returns **false**.
boolean before(Date *date*)	Returns **true** if the invoking **Date** object contains a date that is earlier than the one specified by *date*. Otherwise, it returns **false**.
Object clone( )	Duplicates the invoking **Date** object.
int compareTo(Date *date*)	Compares the value of the invoking object with that of *date*. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than *date*. Returns a positive value if the invoking object is later than *date*.
boolean equals(Object *date*)	Returns **true** if the invoking **Date** object contains the same time and date as the one specified by *date*. Otherwise, it returns **false**.
static Date from(Instant t)	Returns a **Date** object corresponding to the **Instant** object passed in *t*. (Added by JDK 8.)
long getTime( )	Returns the number of milliseconds that have elapsed since January 1, 1970.
int hashCode( )	Returns a hash code for the invoking object.
void setTime(long *time*)	Sets the time and date as specified by *time*, which represents an elapsed time in milliseconds from midnight, January 1, 1970.
Instant toInstant( )	Returns an **Instant** object corresponding to the invoking **Date** object. (Added by JDK 8.)
String toString( )	Converts the invoking **Date** object into a string and returns the result.

The non-deprecated **Date** features do not allow you to obtain the individual components of the date or time. you can only obtain the date and time in terms of milliseconds, in its default string representation as returned by **toString( )**, or (beginning with JDK 8) as an **Instant** object.

// Show date and time using only Date methods.

```
import java.util.Date;
class DateDemo
{       public static void main(String args[])
        {       // Instantiate a Date object
                        Date date = new Date();
                // display time and date using toString()
                        System.out.println(date);
        // Display number of milliseconds since midnight, January 1, 1970 GMT
                        long msec = date.getTime();
        System.out.println("Milliseconds since Jan. 1, 1970 GMT = " + msec);
        }
}
```

Sample output is shown here:

```
Wed Jan 01 11:11:44 CST 2014
Milliseconds since Jan. 1, 1970 GMT = 1388596304803
```

Calendar

- The abstract **Calendar** class provides a set of methods that allows you to convert a time in milliseconds to a number of useful components. Some examples of the type of information that can be provided are year, month, day, hour, minute, and second.
- It is intended that subclasses of **Calendar** will provide the specific functionality to interpret time information according to their own rules. This is one aspect of the Java class library that enables you to write programs that can operate in international environments. An example of such a subclass is **GregorianCalendar**.
- **Calendar** provides no public constructors.
- **Calendar** defines several protected instance variables.
- **areFieldsSet** is a **boolean** that indicates if the time components have been set.
- **fields** is an array of **int**s that holds the components of the time.
- **isSet** is a **boolean** array that indicates if a specific time component has been set.
- **time** is a **long** that holds the current time for this object.
- **isTimeSet** is a **boolean** that indicates if the current time has been set.

NOTE: JDK 8 defines a new date and time API in java.time, which new applications may want to employ.

A sampling of methods defined by **Calendar** are shown in Table

Method	**Description**
abstract void add(int *which*, int *val*)	Adds *val* to the time or date component specified by *which*. To subtract, add a negative value. *which* must be one of the fields defined by **Calendar**, such as **Calendar.HOUR**.
boolean after(Object *calendarObj*)	Returns **true** if the invoking **Calendar** object contains a date that is later than the one specified by *calendarObj*. Otherwise, it returns **false**.
boolean before(Object *calendarObj*)	Returns **true** if the invoking **Calendar** object contains a date that is earlier than the one specified by *calendarObj*. Otherwise, it returns **false**.
final void clear( )	Zeros all time components in the invoking object.
final void clear(int *which*)	Zeros the time component specified by *which* in the invoking object.
Object clone( )	Returns a duplicate of the invoking object.
boolean equals(Object *calendarObj*)	Returns **true** if the invoking **Calendar** object contains a date that is equal to the one specified by *calendarObj*. Otherwise, it returns **false**.
int get(int *calendarField*)	Returns the value of one component of the invoking object. The component is indicated by *calendarField*. Examples of the components that can be requested are **Calendar.YEAR**, **Calendar.MONTH**, **Calendar.MINUTE**, and so forth.
static Locale[ ] getAvailableLocales( )	Returns an array of **Locale** objects that contains the locales for which calendars are available.
static Calendar getInstance( )	Returns a **Calendar** object for the default locale and time zone.
static Calendar getInstance(TimeZone *tz*)	Returns a **Calendar** object for the time zone specified by *tz*. The default locale is used.
static Calendar getInstance(Locale *locale*)	Returns a **Calendar** object for the locale specified by *locale*. The default time zone is used.
static Calendar getInstance(TimeZone *tz*, Locale *locale*)	Returns a **Calendar** object for the time zone specified by *tz* and the locale specified by *locale*.
final Date getTime( )	Returns a **Date** object equivalent to the time of the invoking object.
TimeZone getTimeZone( )	Returns the time zone for the invoking object.
final boolean isSet(int *which*)	Returns **true** if the specified time component is set. Otherwise, it returns **false**.
void set(int *which*, int *val*)	Sets the date or time component specified by *which* to the value specified by *val* in the invoking object. *which* must be one of the fields defined by **Calendar**, such as **Calendar.HOUR**.
final void set(int *year*, int *month*, int *dayOfMonth*)	Sets various date and time components of the invoking object.
final void set(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*)	Sets various date and time components of the invoking object.
final void set(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*, int *seconds*)	Sets various date and time components of the invoking object.
final void setTime(Date *d*)	Sets various date and time components of the invoking object. This information is obtained from the **Date** object *d*.
void setTimeZone(TimeZone *tz*)	Sets the time zone for the invoking object to that specified by *tz*.
final Instant toInstant( )	Returns an **Instant** object corresponding to the invoking **Calendar** instance. (Added by JDK 8.)

Calendar defines the following **int** constants, which are used when you get or set components of the calendar. (The ones with the suffix **FORMAT** or **STANDALONE** were added by JDK 8.)

ALL_STYLES	HOUR_OF_DAY	PM
AM	JANUARY	SATURDAY
AM_PM	JULY	SECOND
APRIL	JUNE	SEPTEMBER
AUGUST	LONG	SHORT
DATE	LONG_FORMAT	SHORT_FORMAT
DAY_OF_MONTH	LONG_STANDALONE	SHORT_STANDALONE
DAY_OF_WEEK	MARCH	SUNDAY
DAY_OF_WEEK_IN_MONTH	MAY	THURSDAY
DAY_OF_YEAR	MILLISECOND	TUESDAY
DECEMBER	MINUTE	UNDECIMBER
DST_OFFSET	MONDAY	WEDNESDAY
ERA	MONTH	WEEK_OF_MONTH
FEBRUARY	NARROW_FORMAT	WEEK_OF_YEAR
FIELD_COUNT	NARROW_STANDALONE	YEAR
FRIDAY	NOVEMBER	ZONE_OFFSET
HOUR	OCTOBER	

```
// Demonstrate Calendar
import java.util.Calendar;
class CalendarDemo
{       public static void main(String args[])
        {       String months[] = {"Jan", "Feb", "Mar", "Apr","May", "Jun", "Jul",
                "Aug", "Sep", "Oct", "Nov", "Dec"};
                // Create a calendar initialized with the current date and time in the
                //default  locale and timezone.
                        Calendar calendar = Calendar.getInstance();
                // Display current time and date information.
                        System.out.print("Date: ");
                        System.out.print(months[calendar.get(Calendar.MONTH)]);
                        System.out.print(" " + calendar.get(Calendar.DATE) + " ");
                        System.out.println(calendar.get(Calendar.YEAR));
                        System.out.print("Time: ");
                        System.out.print(calendar.get(Calendar.HOUR) + ":");
                        System.out.print(calendar.get(Calendar.MINUTE) + ":");
                        System.out.println(calendar.get(Calendar.SECOND));
                // Set the time and date information and display it.
```

```
                        calendar.set(Calendar.HOUR, 10);
                        calendar.set(Calendar.MINUTE, 29);
                        calendar.set(Calendar.SECOND, 22);
                        System.out.print("Updated time: ");
                        System.out.print(calendar.get(Calendar.HOUR) + ":");
                        System.out.print(calendar.get(Calendar.MINUTE) + ":");
                        System.out.println(calendar.get(Calendar.SECOND));
        }
}
```

Sample output is shown here:

```
Date: Jan 1 2014
Time: 11:29:39
Updated time: 10:29:22
```

GregorianCalendar

- **GregorianCalendar** is a concrete implementation of a **Calendar** that implements the normal Gregorian calendar with which you are familiar.
- The **getInstance( )** method of **Calendar** will typically return a G**regorianCalendar** initialized with the current date and time in the default locale and time zone.
- **GregorianCalendar** defines two fields: **AD** and **BC**. These represent the two eras defined by the Gregorian calendar.

There are also several constructors for **GregorianCalendar** objects.
The default, **GregorianCalendar( )**, initializes the object with the current date and time in the default locale and time zone.
Three more constructors offer increasing levels of specificity:
GregorianCalendar(int *year*, int *month*, int *dayOfMonth*): sets the time to midnight
GregorianCalendar(int *year*, int *month*, int *dayOfMonth*, int *hours*,
    int *minutes*): sets the hours and the minutes.
GregorianCalendar(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*, int
    seconds): adds seconds

All three versions set the day, month, and year. Here, *year* specifies the year. The month is specified by *month*, with zero indicating January. The day of the month is specified by *dayOfMonth*.

You can also construct a **GregorianCalendar** object by specifying the locale and/or time zone. The following constructors create objects initialized with the current date and time using the specified time zone and/or locale:
GregorianCalendar(Locale *locale*)

GregorianCalendar(TimeZone *timeZone*)
GregorianCalendar(TimeZone *timeZone*, Locale *locale*)

GregorianCalendar provides an implementation of all the abstract methods in **Calendar**. It also provides some additional methods.
isLeapYear( ): tests if the year is a leap year. Its form is boolean isLeapYear(int *year*): returns **true** if *year* is a leap year and **false** otherwise.

JDK 8 also adds the following methods: **from( )** and **toZonedDateTime( )**, which support the new date and time API, and **getCalendarType( )**, which returns the calendar type as a string, which is "gregory".

The following program demonstrates **GregorianCalendar**:

```
// Demonstrate GregorianCalendar
import java.util.*;
class GregorianCalendarDemo
{       public static void main(String args[])
        {       String months[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
"Aug", "Sep", "Oct", "Nov", "Dec"};
                int year;
                // Create a Gregorian calendar initialized with the current date and
                //time in the default locale and timezone.
                GregorianCalendar gcalendar = new GregorianCalendar();
                // Display current time and date information.
                        System.out.print("Date: ");
                        System.out.print(months[gcalendar.get(Calendar.MONTH)]);
                        System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
                        System.out.println(year = gcalendar.get(Calendar.YEAR));
                        System.out.print("Time: ");
                        System.out.print(gcalendar.get(Calendar.HOUR) + ":");
                        System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
                        System.out.println(gcalendar.get(Calendar.SECOND));
                // Test if the current year is a leap year
                        if(gcalendar.isLeapYear(year))
                        {       System.out.println("The current year is a leap year");
                        }
                        else
                        {       System.out.println("The current year is not a leap year");
                        }
        }
}
```

Sample output is shown here:

Date: Jan 1 2014
Time: 1:45:5
The current year is not a leap year

Random

- The **Random** class is a generator of pseudorandom numbers.
- These are called *pseudorandom* numbers because they are simply uniformly distributed sequences.

Random defines the following constructors:
Random( ): creates a number generator that uses a reasonably unique seed.
Random(long *seed*): specify a seed value manually.

If you initialize a **Random** object with a seed, you define the starting point for the random sequence. If you use the same seed to initialize another **Random** object, you will extract the same random sequence. If you want to generate different sequences, specify different seed values. One way to do this is to use the current time to seed a **Random** object. This approach reduces the possibility of getting repeated sequences.

The core public methods defined by **Random** are shown in Table.

Method	Description
boolean nextBoolean( )	Returns the next **boolean** random number.
void nextBytes(byte *vals*[ ])	Fills *vals* with randomly generated values.
double nextDouble( )	Returns the next **double** random number.
float nextFloat( )	Returns the next **float** random number.
double nextGaussian( )	Returns the next Gaussian random number.
int nextInt( )	Returns the next **int** random number.
int nextInt(int *n*)	Returns the next **int** random number within the range zero to *n*.
long nextLong( )	Returns the next **long** random number.
void setSeed(long *newSeed*)	Sets the seed value (that is, the starting point for the random number generator) to that specified by *newSeed*.

There are seven types of random numbers that you can extract from a **Random** object.

- Random Boolean values are available from **nextBoolean( )**.
- Random bytes can be obtained by calling **nextBytes( )**.
- Integers can be extracted via the **nextInt( )** method.
- Long integers, uniformly distributed over their range, can be obtained with **nextLong( )**.
- The **nextFloat( )** and **nextDouble( )** methods return a uniformly distributed **float** and **double**, respectively, between 0.0 and 1.0.

- **nextGaussian( )** returns a **double** value centered at 0.0 with a standard deviation of 1.0. This is what is known as a *bell curve*.

```
// Demonstrate random Gaussian values.
import java.util.Random;
class RandDemo
{       public static void main(String args[])
        {       Random r = new Random();
                double val;
                double sum = 0;
                int bell[] = new int[10];
                for(int i=0; i<100; i++)
                {       val = r.nextGaussian();
                        sum += val;
                        double t = -2;
                        for(int x=0; x<10; x++, t += 0.5)
                        if(val < t)
                        {       bell[x]++;
                                break;
                        }
                }
                System.out.println("Average of values: " +(sum/100));
                // display bell curve, sideways
                for(int i=0; i<10; i++)
                {       for(int x=bell[i]; x>0; x--)
                                System.out.print("*");
                        System.out.println();
                }
        }
}
```

a bell-like distribution of numbers is obtained.

```
Average of values: 0.0702235271133344
**
*******
******
***************
******************
*****************
*************
**********
********
***
```

JDK 8 adds three new methods to **Random** that support the new stream API.

They are called **doubles( )**, **ints( )**, and **longs( )**, and each returns a reference to a stream that contains a sequence of pseudorandom values of the specified type. Each method defines several overloads. Here are their simplest forms:

DoubleStream doubles( ): returns a stream that contains pseudorandom **double** values. (The range of these values will be less than 1.0 but greater than 0.0.)

IntStream ints( ): returns a stream that contains pseudorandom **int** values

LongStream longs( ): returns a stream that contains pseudorandom **long** values.

For these three methods, the stream returned is effectively infinite. Several overloads of each method are provided that let you specify the size of the stream, an origin, and an upper bound.

Formatter

- For creating formatted output, **Formatter** class is used.
- It provides *format conversions* that let you display numbers, strings, and time and date in virtually any format you like.
- It operates in a manner similar to the C/C++ **printf( )** function
- It also further streamlines the conversion of C/C++ code to Java.

NOTE Although Java's Formatter class operates in a manner very similar to the C/C++ printf( ) function, there are some differences, and some new features.

The Formatter Constructors

Before you can use **Formatter** to format output, you must create a **Formatter** object. In general, **Formatter** works by converting the binary form of data used by a program into formatted text. It stores the formatted text in a buffer, the contents of which can be obtained by your program whenever they are needed. It is possible to let **Formatter** supply this buffer automatically, or you can specify the buffer explicitly when a **Formatter** object is created. It is also possible to have **Formatter** output its buffer to a file.

The **Formatter** class defines many constructors:

Formatter( )
Formatter(Appendable *buf*)
Formatter(Appendable *buf*, Locale *loc*)
Formatter(String *filename*)
throws FileNotFoundException
Formatter(String *filename*, String *charset*)
throws FileNotFoundException, UnsupportedEncodingException

Formatter(File *outF*)
throws FileNotFoundException
Formatter(OutputStream *outStrm*)

Here, *buf* specifies a buffer for the formatted output. If *buf* is null, then **Formatter** automatically allocates a **StringBuilder** to hold the formatted output.

The *loc* parameter specifies a locale. If no locale is specified, the default locale is used.

The *filename* parameter specifies the name of a file that will receive the formatted output.

The *charset* parameter specifies the character set. If no character set is specified, then the default character set is used.

The *outF* parameter specifies a reference to an open file that will receive output.

The *outStrm* parameter specifies a reference to an output stream that will receive output. When using a file, output is also written to the file.

Perhaps the most widely used constructor is the first, which has no parameters. It automatically uses the default locale and allocates a **StringBuilder** to hold the formatted output.

The Formatter Methods

Formatter defines the methods shown in Table

Method	Description
void close( )	Closes the invoking **Formatter**. This causes any resources used by the object to be released. After a **Formatter** has been closed, it cannot be reused. An attempt to use a closed **Formatter** results in a **FormatterClosedException**.
void flush( )	Flushes the format buffer. This causes any output currently in the buffer to be written to the destination. This applies mostly to a **Formatter** tied to a file.
Formatter format(String *fmtString*, Object ... *args*)	Formats the arguments passed via *args* according to the format specifiers contained in *fmtString*. Returns the invoking object.
Formatter format(Locale *loc*, String *fmtString*, Object ... *args*)	Formats the arguments passed via *args* according to the format specifiers contained in *fmtString*. The locale specified by *loc* is used for this format. Returns the invoking object.
IOException ioException( )	If the underlying object that is the destination for output throws an **IOException**, then this exception is returned. Otherwise, null is returned.
Locale locale( )	Returns the invoking object's locale.
Appendable out( )	Returns a reference to the underlying object that is the destination for output.
String toString( )	Returns a **String** containing the formatted output.

Formatting Basics

To create a formatted string we use **format( )** method after creating **Formatter**

The most commonly used version is shown here:

Formatter format(String *fmtString*, Object ... *args*)

The *fmtSring* consists of two types of items.

The first type is composed of characters that are simply copied to the output buffer.

The second type contains *format specifiers* that define the way the subsequent arguments are displayed.

In its simplest form, a format specifier begins with a percent sign followed by the format *conversion specifier* Ex:**%f**.

The **format( )** method accepts a wide variety of format specifiers

Format Specifier	Conversion Applied
%a %A	Floating-point hexadecimal
%b %B	Boolean
%c	Character
%d	Decimal integer
%h %H	Hash code of the argument
%e %E	Scientific notation
%f	Decimal floating-point
%g %G	Uses %e or %f, based on the value being formatted and the precision
%o	Octal integer
%n	Inserts a newline character
%s %S	String
%t %T	Time and date
%x %X	Integer hexadecimal
%%	Inserts a % sign

If the argument doesn't match, an **IllegalFormatException** is thrown.

Once you have formatted a string, you can obtain it by calling **toString( )**.

```
        String str = fmt.toString();
```

```
// A very simple example that uses Formatter.
import java.util.*;
class FormatDemo
{       public static void main(String args[])
```

```
        {       Formatter fmt = new Formatter();
                fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.6);
                System.out.println(fmt);
                fmt.close();
        }
}
```

Formatting Strings and Characters

To format an individual character, use **%c**

To format a string, use **%s**.

Formatting Numbers

To format an integer in decimal format, use **%d**.

To format a floating-point value in decimal format, use **%f**.

To format a floating-point value in scientific notation, use **%e**. Numbers represented in scientific notation take this general form: *x.dddddd*e+/–yy

The **%g** format specifier causes **Formatter** to use either **%f** or **%e**, based on the value being formatted and the precision, which is 6 by default.

```
// Demonstrate the %f and %e format specifiers.
import java.util.*;
class FormatDemo2
{       public static void main(String args[])
        {       Formatter fmt = new Formatter();
                for(double i=1.23; i < 1.0e+6; i *= 100)
                {       fmt.format("%f %e", i, i);
                        System.out.println(fmt);
                }
                fmt.close();
        }
}
```

It produces the following output:

```
1.230000 1.230000e+00
1.230000 1.230000e+00 123.000000 1.230000e+02
1.230000 1.230000e+00 123.000000 1.230000e+02 12300.000000 1.230000e+04
```

Formatting Time and Date

The **%t** specifier works a bit differently than the others because it requires the use of a suffix to describe the portion and precise format of the time or date desired. The suffixes are shown in Table

Suffix	Replaced By
a	Abbreviated weekday name
A	Full weekday name
b	Abbreviated month name
B	Full month name
c	Standard date and time string formatted as *day month date hh::mm:ss tzone year*
C	First two digits of year
d	Day of month as a decimal (01—31)
D	month/day/year
e	Day of month as a decimal (1—31)
F	year-month-day
h	Abbreviated month name
H	Hour (00 to 23)
I	Hour (01 to 12)
j	Day of year as a decimal (001 to 366)
k	Hour (0 to 23)
l	Hour (1 to 12)
L	Millisecond (000 to 999)
m	Month as decimal (01 to 13)
M	Minute as decimal (00 to 59)
N	Nanosecond (000000000 to 999999999)
p	Locale's equivalent of AM or PM in lowercase
Q	Milliseconds from 1/1/1970
r	*hh:mm:ss* (12-hour format)
R	*hh:mm* (24-hour format)
S	Seconds (00 to 60)
s	Seconds from 1/1/1970 UTC
T	*hh:mm:ss* (24-hour format)
y	Year in decimal without century (00 to 99)
Y	Year in decimal including century (0001 to 9999)
z	Offset from UTC
Z	Time zone name

For example, to display minutes, you would use **%tM**, where **M** indicates minutes in a two-character field.

```
// Formatting time and date.
import java.util.*;
class TimeDateFormat
{       public static void main(String args[])
        {       Formatter fmt = new Formatter();
```

```
                Calendar cal = Calendar.getInstance();
                // Display standard 12-hour time format.
                        fmt.format("%tr", cal);
                        System.out.println(fmt);
                        fmt.close();
                // Display complete time and date information.
                        fmt = new Formatter();
                        fmt.format("%tc",  cal);
                        System.out.println(fmt);
                        fmt.close();
                // Display just hour and minute.
                        fmt = new Formatter();
                        fmt.format("%tl:%tM", cal, cal);
                        System.out.println(fmt);
                        fmt.close();
                // Display month by name and number.
                        fmt = new Formatter();
                        fmt.format("%tB %tb %tm", cal, cal, cal);
                        System.out.println(fmt);
                        fmt.close();
        }
}
```

Sample output is shown here:

```
03:15:34 PM
Wed Jan 01 15:15:34 CST 2014
3:15
January Jan 01
```

```
// Demonstrate the %n and %% format specifiers.
import java.util.*;
class FormatDemo3
{       public static void main(String args[])
        {       Formatter fmt = new Formatter();
                fmt.format("Copying file%nTransfer is %d%% complete", 88);
                System.out.println(fmt);
                fmt.close();
        }
}
```

It displays the following output:

```
Copying file
Transfer is 88% complete
```

Specifying a Minimum Field Width

An integer placed between the **%** sign and the format conversion code acts as a *minimum field-width specifier*. This pads the output with spaces to ensure that it reaches a certain minimum length. If the string or number is longer than that minimum, it will still be printed in full. The default padding is done with spaces. If you want to pad with 0's, place a 0 before the field-width specifier. For example, **%05d** will pad a number of less than five digits with 0's so that its total length is five. The field-width specifier can be used with all format specifiers except **%n**.

```
// Demonstrate a field-width specifier.
import java.util.*;
class FormatDemo4
{	public static void main(String args[])
	{	Formatter fmt = new Formatter();
		fmt.format("|%f|%n|%12f|%n|%012f|",10.12345,10.12345, 10.12345);
		System.out.println(fmt);
		fmt.close();
	}
}
```

This program produces the following output:

```
|10.123450|
| 10.123450|
|00010.123450|
```

```
// Create a table of squares and cubes.
import java.util.*;
class FieldWidthDemo
{	public static void main(String args[])
	{	Formatter fmt;
		for(int i=1; i <= 10; i++)
		{	fmt = new Formatter();
			fmt.format("%4d %4d %4d", i, i*i, i*i*i);
			System.out.println(fmt);
			fmt.close();
		}
	}
}
```

Its output is shown here:

```
1 1 1
2 4 8
3 9 27
4 16 64
```

```
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

Specifying Precision

A *precision specifier* can be applied to the **%f**, **%e**, **%g**, and **%s** format specifiers. It follows the minimum field-width specifier (if there is one) and consists of a period followed by an integer. Its exact meaning depends upon the type of data to which it is applied. When you apply the precision specifier to floating-point data using the **%f** or **%e** specifiers, it determines the number of decimal places displayed. For example, **%10.4f** displays a number at least ten characters wide with four decimal places. When using **%g**, the precision determines the number of significant digits. The default precision is 6.

Applied to strings, the precision specifier specifies the maximum field length. For example, **%5.7s** displays a string of at least five and not exceeding seven characters long. If the string is longer than the maximum field width, the end characters will be truncated.

```
// Demonstrate the precision modifier.
import java.util.*;
class PrecisionDemo
{       public static void main(String args[])
        {       Formatter fmt = new Formatter();
                // Format 4 decimal places.
                        fmt.format("%.4f", 123.1234567);
                System.out.println(fmt);
                fmt.close();
                // Format to 2 decimal places in a 16 character field
                        fmt = new Formatter();
                        fmt.format("%16.2e", 123.1234567);
                System.out.println(fmt);
                fmt.close();
                // Display at most 15 characters in a string.
                        fmt = new Formatter();
                        fmt.format("%.15s", "Formatting with Java is now easy.");
                        System.out.println(fmt);
                        fmt.close();
        }
}
```

It produces the following output:

```
123.1235
1.23e+02
Formatting with
```

Using the Format Flags

Formatter recognizes a set of format *flags* that lets you control various aspects of a conversion. All format flags are single characters, and a format flag follows the **%** in a format specification. The flags are shown here:

Flag	Effect
–	Left justification
#	Alternate conversion format
0	Output is padded with zeros rather than spaces
space	Positive numeric output is preceded by a space
+	Positive numeric output is preceded by a + sign
,	Numeric values include grouping separators
(	Negative numeric values are enclosed within parentheses

Not all flags apply to all format specifiers.

Justifying Output

By default, all output is right-justified. That is, if the field width is larger than the data printed, the data will be placed on the right edge of the field. You can force output to be left-justified by placing a minus sign directly after the %. For instance, **%–10.2f** left-justifies a floating-point number with two decimal places in a 10-character field

```
// Demonstrate left justification.
import java.util.*;
class LeftJustify
{       public static void main(String args[])
        {       Formatter fmt = new Formatter();
                // Right justify by default
                        fmt.format("|%10.2f|", 123.123);
                        System.out.println(fmt);
                        fmt.close();
                // Now, left justify.
                        fmt = new Formatter();
                        fmt.format("|%-10.2f|", 123.123);
                        System.out.println(fmt);
                        fmt.close();
```

```
	}
}
```

It produces the following output:

```
| 123.12|
|123.12 |
```

The Space, +, 0, and (Flags

To cause a + sign to be shown before positive numeric values, add the + flag. For example, fmt.format("%+d", 100); creates this string: +100

When creating columns of numbers, it is sometimes useful to output a space before positive values so that positive and negative values line up. To do this, add the space flag.

```
// Demonstrate the space format specifiers.
import java.util.*;
class FormatDemo5
{	public static void main(String args[])
	{	Formatter fmt = new Formatter();
		fmt.format("% d", -100);
		System.out.println(fmt);
		fmt.close();
		fmt = new Formatter();
		fmt.format("% d", 100);
		System.out.println(fmt);
		fmt.close();
		fmt = new Formatter();
		fmt.format("% d", -200);
		System.out.println(fmt);
		fmt.close();
		fmt = new Formatter();
		fmt.format("% d", 200);
		System.out.println(fmt);
		fmt.close();
	}
}
```

The output is shown here:

```
-100
100
-200
200
```

Notice that the positive values have a leading space, which causes the digits in the column To show negative numeric output inside parentheses, rather than with a leading –, use the **(** flag. For example, fmt.format("%(d", -100);

creates this string: (100)

The 0 flag causes output to be padded with zeros rather than spaces.

The Comma Flag

When displaying large numbers, it is often useful to add grouping separators, which in English are commas. For example, the value 1234567 is more easily read when formatted as 1,234,567. To add grouping specifiers, use the comma **(,)** flag. For example, fmt.format("%,.2f", 4356783497.34);

creates this string:4,356,783,497.34

The # Flag

The # can be applied to **%o**, **%x**, **%e,** and **%f**. For **%e,** and **%f**, the # ensures that there will be a decimal point even if there are no decimal digits. If you precede the **%x** format specifier with a #, the hexadecimal number will be printed with a **0x** prefix. Preceding the **%o** specifier with # causes the number to be printed with a leading zero.

The Uppercase Option

Several of the format specifiers have uppercase versions that cause the conversion to use uppercase where appropriate. The following table describes the effect.

Specifier	Effect
%A	Causes the hexadecimal digits *a* through *f* to be displayed in uppercase as *A* through *F.* Also, the prefix **0x** is displayed as **0X**, and the **p** will be displayed as **P**.
%B	Uppercases the values **true** and **false**.
%E	Causes the *e* symbol that indicates the exponent to be displayed in uppercase.
%G	Causes the *e* symbol that indicates the exponent to be displayed in uppercase.
%H	Causes the hexadecimal digits *a* through *f* to be displayed in uppercase as *A* through *F.*
%S	Uppercases the corresponding string.
%T	Causes all alphabetical output to be displayed in uppercase.
%X	Causes the hexadecimal digits *a* through *f* to be displayed in uppercase as *A* through *F.* Also, the optional prefix **0x** is displayed as **0X**, if present.

Using an Argument Index

Formatter includes a very useful feature that lets you specify the argument to which a format specifier applies. Normally, format specifiers and arguments are matched in order, from left to right. That is, the first format specifier matches the first argument, the second format specifier matches the second argument, and so on. However, by using an *argument index*, you can explicitly control which argument a format specifier matches. An argument index immediately follows the **%** in a format specifier. It has the following format: *n*$

where *n* is the index of the desired argument, beginning with 1.

For example, consider this example:

```
fmt.format("%3$d %1$d %2$d", 10, 20, 30);
```

It produces this string:

```
30 10 20
```

One advantage of argument indexes is that they enable you to reuse an argument without having to specify it twice.

For example, consider this line:

```
fmt.format("%d in hex is %1$x", 255);
```

It produces the following string:

```
255 in hex is ff
```

There is a convenient shorthand called a *relative index* that enables you to reuse the argument matched by the preceding format specifier. Simply specify **<** for the argument index. For example, the following call to **format( )** produces the same results as the previous example:

```
fmt.format("%d in hex is %<x", 255);
```

Relative indexes are especially useful when creating custom time and date formats.

```
// Use relative indexes to simplify the creation of a custom time and date format.
import java.util.*;
class FormatDemo6
{       public static void main(String args[])
        {       Formatter fmt = new Formatter();
                Calendar cal = Calendar.getInstance();
                fmt.format("Today is day %te of %<tB, %<tY", cal);
                System.out.println(fmt);
                fmt.close();
        }
}
```

Here is sample output:

```
Today is day 1 of January, 2014
```

Because of relative indexing, the argument **cal** need only be passed once, rather than three times.

Closing a Formatter

In general, you should close a **Formatter** when you are done using it. Doing so frees any resources that it was using. This is especially important when formatting to a file, but it can be important in other cases, too. As the previous examples have shown, one way to close a **Formatter** is to explicitly call **close( )**. However, beginning with JDK 7, **Formatter** implements the **AutoCloseable** interface. This means that it supports the **try**-with-resources statement. Using this approach, the **Formatter** is automatically closed when it is no longer needed.

```
// Use automatic resource management with Formatter.
import java.util.*;
class FormatDemo
{       public static void main(String args[])
        {       try (Formatter fmt = new Formatter())
                {       fmt.format("Formatting %s is easy %d %f", "with Java", 10,
                        98.6);
                        System.out.println(fmt);
                }
        }
}
```

The output is the same as before.

The Java printf( ) Connection

The **printf( )** method automatically uses **Formatter** to create a formatted string. It then displays that string on **System.out**, which is the console by default. The **printf( )** method is defined by both **PrintStream** and **PrintWriter**.

Scanner

- **Scanner** is the complement of **Formatter**.
- It reads formatted input and converts it into its binary form.
- **Scanner** can be used to read input from the console, a file, a string, or any source that implements the **Readable** interface or **ReadableByteChannel**.
- In general, a **Scanner** can be created for a **String**, an **InputStream**, a **File**, or any object that implements the **Readable** or **ReadableByteChannel** interfaces.

The Scanner Constructors

Scanner defines the constructors shown in Table

Method	Description
Scanner(File *from*) throws FileNotFoundException	Creates a **Scanner** that uses the file specified by *from* as a source for input.
Scanner(File *from*, String *charset*) throws FileNotFoundException	Creates a **Scanner** that uses the file specified by *from* with the encoding specified by *charset* as a source for input.
Scanner(InputStream *from*)	Creates a **Scanner** that uses the stream specified by *from* as a source for input.
Scanner(InputStream *from*, String *charset*)	Creates a **Scanner** that uses the stream specified by *from* with the encoding specified by *charset* as a source for input.
Scanner(Path *from*) throws IOException	Creates a **Scanner** that uses the file specified by *from* as a source for input.
Scanner(Path *from*, String *charset*) throws IOException	Creates a **Scanner** that uses the file specified by *from* with the encoding specified by *charset* as a source for input.
Scanner(Readable *from*)	Creates a **Scanner** that uses the **Readable** object specified by *from* as a source for input.
Scanner (ReadableByteChannel *from*)	Creates a **Scanner** that uses the **ReadableByteChannel** specified by *from* as a source for input.
Scanner(ReadableByteChannel *from*, String *charset*)	Creates a **Scanner** that uses the **ReadableByteChannel** specified by *from* with the encoding specified by *charset* as a source for input.
Scanner(String *from*)	Creates a **Scanner** that uses the string specified by *from* as a source for input.

The following sequence creates a **Scanner** that reads the file **Test.txt**:

```
FileReader fin = new FileReader("Test.txt");
Scanner src = new Scanner(fin);
```

This works because **FileReader** implements the **Readable** interface. Thus, the call to the constructor resolves to **Scanner(Readable)**.

This next line creates a **Scanner** that reads from standard input, which is the keyboard by default:

```
Scanner conin = new Scanner(System.in);
```

This works because **System.in** is an object of type **InputStream**. Thus, the call to the constructor maps to **Scanner(InputStream)**.

The next sequence creates a **Scanner** that reads from a string.

```
String instr = "10 99.88 scanning is easy.";
Scanner conin = new Scanner(instr);
```

Scanning Basics

- A **Scanner** reads *tokens* from the underlying source that you specified when the **Scanner** was created.
- As it relates to **Scanner**, a token is a portion of input that is delineated by a set of delimiters, which is whitespace by default.
- A token is read by matching it with a particular *regular expression*, which defines the format of the data. Although **Scanner** allows you to define the specific type of expression that its next input operation will match, it

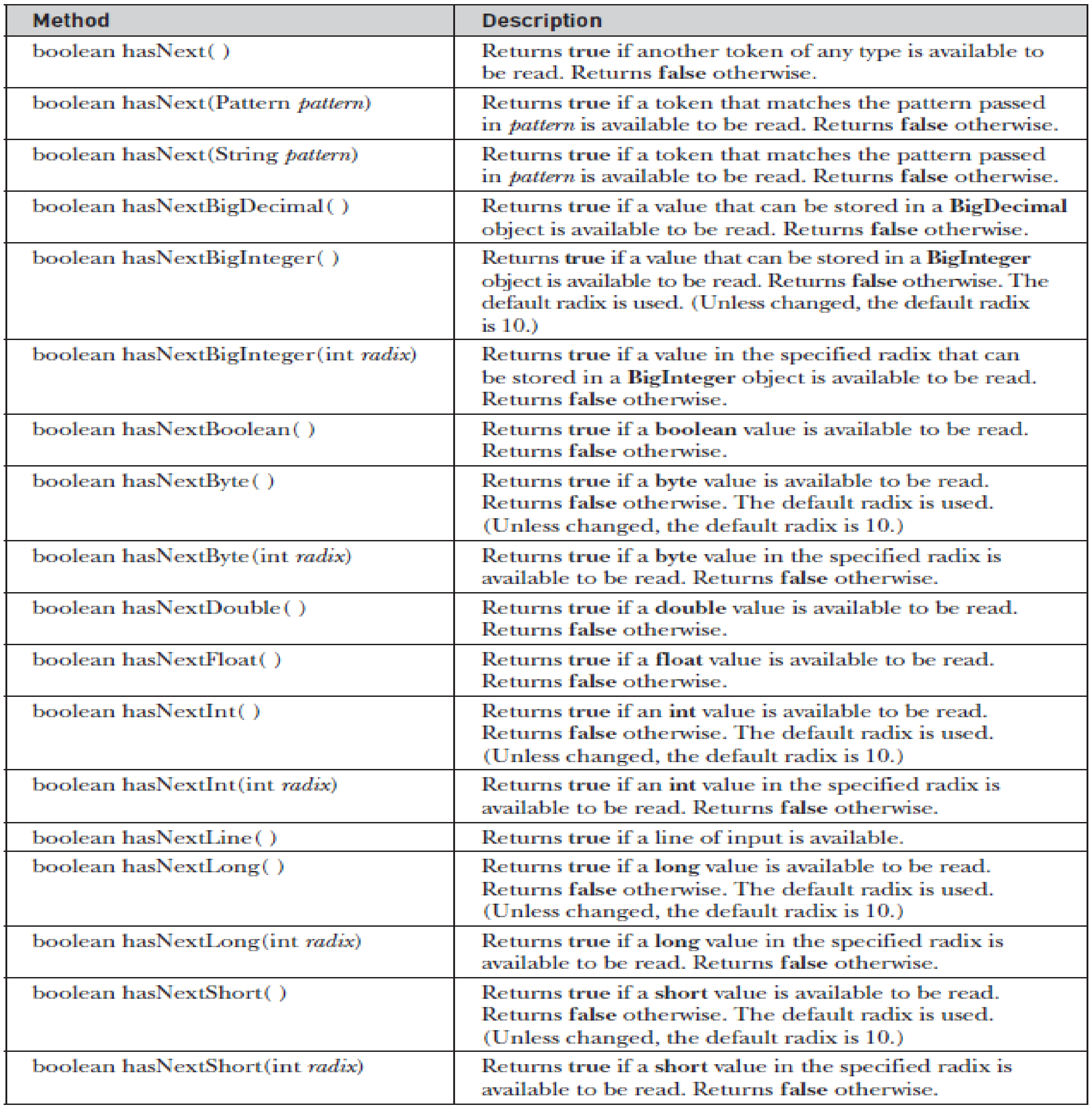
includes many predefined patterns, which match the primitive types, such as **int** and **double**,and strings. Thus, often you won't need to specify a pattern to match.

In general, to use **Scanner**, follow this procedure:

1. Determine if a specific type of input is available by calling one of **Scanner**'s **hasNext***X* methods, where *X* is the type of data desired.
2. If input is available, read it by calling one of **Scanner**'s **next***X* methods.
3. Repeat the process until input is exhausted.
4. Close the **Scanner** by calling **close( )**.

Scanner defines two sets of methods that enable you to read input.

- The first are the **hasNext***X* methods, which are shown in Table



Method	Description
boolean hasNext( )	Returns **true** if another token of any type is available to be read. Returns **false** otherwise.
boolean hasNext(Pattern *pattern*)	Returns **true** if a token that matches the pattern passed in *pattern* is available to be read. Returns **false** otherwise.
boolean hasNext(String *pattern*)	Returns **true** if a token that matches the pattern passed in *pattern* is available to be read. Returns **false** otherwise.
boolean hasNextBigDecimal( )	Returns **true** if a value that can be stored in a **BigDecimal** object is available to be read. Returns **false** otherwise.
boolean hasNextBigInteger( )	Returns **true** if a value that can be stored in a **BigInteger** object is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextBigInteger(int *radix*)	Returns **true** if a value in the specified radix that can be stored in a **BigInteger** object is available to be read. Returns **false** otherwise.
boolean hasNextBoolean( )	Returns **true** if a **boolean** value is available to be read. Returns **false** otherwise.
boolean hasNextByte( )	Returns **true** if a **byte** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextByte(int *radix*)	Returns **true** if a **byte** value in the specified radix is available to be read. Returns **false** otherwise.
boolean hasNextDouble( )	Returns **true** if a **double** value is available to be read. Returns **false** otherwise.
boolean hasNextFloat( )	Returns **true** if a **float** value is available to be read. Returns **false** otherwise.
boolean hasNextInt( )	Returns **true** if an **int** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextInt(int *radix*)	Returns **true** if an **int** value in the specified radix is available to be read. Returns **false** otherwise.
boolean hasNextLine( )	Returns **true** if a line of input is available.
boolean hasNextLong( )	Returns **true** if a **long** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextLong(int *radix*)	Returns **true** if a **long** value in the specified radix is available to be read. Returns **false** otherwise.
boolean hasNextShort( )	Returns **true** if a **short** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextShort(int *radix*)	Returns **true** if a **short** value in the specified radix is available to be read. Returns **false** otherwise.

If the desired data is available, then you read it by calling one of **Scanner**'s **next***X* methods, which are shown in Table

Method	Description
String next( )	Returns the next token of any type from the input source.
String next(Pattern *pattern*)	Returns the next token that matches the pattern passed in *pattern* from the input source.
String next(String *pattern*)	Returns the next token that matches the pattern passed in *pattern* from the input source.
BigDecimal nextBigDecimal( )	Returns the next token as a **BigDecimal** object.
BigInteger nextBigInteger( )	Returns the next token as a **BigInteger** object. The default radix is used. (Unless changed, the default radix is 10.)
BigInteger nextBigInteger(int *radix*)	Returns the next token (using the specified radix) as a **BigInteger** object.
boolean nextBoolean( )	Returns the next token as a **boolean** value.
byte nextByte( )	Returns the next token as a **byte** value. The default radix is used. (Unless changed, the default radix is 10.)
byte nextByte(int *radix*)	Returns the next token (using the specified radix) as a **byte** value.
double nextDouble( )	Returns the next token as a **double** value.
float nextFloat( )	Returns the next token as a **float** value.
int nextInt( )	Returns the next token as an **int** value. The default radix is used. (Unless changed, the default radix is 10.)
int nextInt(int *radix*)	Returns the next token (using the specified radix) as an **int** value.
String nextLine( )	Returns the next line of input as a string.
long nextLong( )	Returns the next token as a **long** value. The default radix is used. (Unless changed, the default radix is 10.)
long nextLong(int *radix*)	Returns the next token (using the specified radix) as a **long** value.
short nextShort( )	Returns the next token as a **short** value. The default radix is used. (Unless changed, the default radix is 10.)
short nextShort(int *radix*)	Returns the next token (using the specified radix) as a **short** value.

The following sequence shows how to read a list of integers from the keyboard.

```
Scanner conin = new Scanner(System.in);
int i;
// Read a list of integers.
while(conin.hasNextInt())
{       i = conin.nextInt();
        // ...
}
```

If a **next** method cannot find the type of data it is looking for, it throws an **InputMismatchException**. A **NoSuchElementException** is thrown if no more input is available

Some Scanner Examples

Scanner makes what could be a tedious task into an easy one.

```
// Use Scanner to compute an average of the values.
import java.util.*;
class AvgNums
{	public static void main(String args[])
	{	Scanner conin = new Scanner(System.in);
		int count = 0;
		double sum = 0.0;
		System.out.println("Enter numbers to average.");
		// Read and sum numbers.
			while(conin.hasNext())
			{	if(conin.hasNextDouble())
				{	sum += conin.nextDouble();
					count++;
				}
				else
				{	String str = conin.next();
					if(str.equals("done"))
						break;
					else
					{	System.out.println("Data format error.");
						return;
					}
				}
			}
			conin.close();
			System.out.println("Average is " + sum / count);
	}
}
```

```
Enter numbers to average.
1.2
2
3.4
4
done
Average is 2.65
```

One thing that is especially nice about **Scanner** is that the same technique used to read from one source can be used to read from another

```
// Use Scanner to compute an average of the values in a file.
import java.util.*;
import java.io.*;
class AvgFile
{       public static void main(String args[])  throws IOException
        {       int count = 0;
                double sum = 0.0;
                // Write output to a file.
                        FileWriter fout = new FileWriter("test.txt");
                        fout.write("2 3.4 5 6 7.4 9.1 10.5 done");
                fout.close();
                FileReader fin = new FileReader("Test.txt");
                Scanner src = new Scanner(fin);
                // Read and sum numbers.
                        while(src.hasNext())
                        {       if(src.hasNextDouble())
                                {       sum += src.nextDouble();
                                        count++;
                                }
                                else
                                {       String str = src.next();
                                        if(str.equals("done"))
                                                break;
                                        else
                                        {       System.out.println("File format error.");
                                                return;
                                        }
                                }
                        }
                        src.close();
                        System.out.println("Average is " + sum / count);
        }
}
```

Here is the output:

```
Average is 6.2
```

When you close a **Scanner**, the **Readable** associated with it is also closed (if that **Readable** implements the **Closeable** interface). Therefore, in this case, the file referred to by **fin** is automatically closed when **src** is closed.

Beginning with JDK 7, **Scanner** also implements the **AutoCloseable** interface. This means that it can be managed by a **try**-with-resources block. When **try**-with-resources is used, the scanner is automatically closed when the block ends.

For example, **src** in the preceding program could have been managed like this:

```
	try (Scanner src = new Scanner(fin))
	{	// Read and sum numbers.
		while(src.hasNext())
		{	if(src.hasNextDouble())
			{	sum += src.nextDouble();
				count++;
			}
			else
			{	String str = src.next();
				if(str.equals("done"))
					break;
				else
				{	System.out.println("File format error.");
					return;
				}
			}
		}
	}
```

One other point: To keep this and the other examples in this section compact, I/O exceptions are simply thrown out of **main( )**. However, your real-world code will normally handle I/O exceptions itself.

You can use **Scanner** to read input that contains several different types of data—even if the order of that data is unknown in advance. You must simply check what type of data is available before reading it. For example, consider this program:

```
// Use Scanner to read various types of data from a file.
import java.util.*;
import java.io.*;
class ScanMixed
{	public static void main(String args[])	throws IOException
	{	int i;
		double d;
		boolean b;
		String str;
		// Write output to a file.
```

```
			FileWriter fout = new FileWriter("test.txt");
			fout.write("Testing Scanner 10 12.2 one true two false");
			fout.close();
			FileReader fin = new FileReader("Test.txt");
			Scanner src = new Scanner(fin);
		// Read to end.
			while(src.hasNext())
			{	if(src.hasNextInt())
				{	i = src.nextInt();
					System.out.println("int: " + i);
				}
				else if(src.hasNextDouble())
				{	d = src.nextDouble();
					System.out.println("double: " + d);
				}
				else if(src.hasNextBoolean())
				{	b = src.nextBoolean();
					System.out.println("boolean: " + b);
				}
				else
				{	str = src.next();
					System.out.println("String: " + str);
				}
			}
			src.close();
		}
}
```

Here is the output:

```
String: Testing
String: Scanner
int: 10
double: 12.2
String: one
boolean: true
String: two
boolean: false
```

When reading mixed data types, as the preceding program does, you need to be a bit careful about the order in which you call the **next** methods. For example, if the loop reversed the order of the calls to **nextInt( )** and **nextDouble( )**, both numeric values would have been read as **double**s, because **nextDouble( )** matches any numeric string that can be represented as a **double**.

Setting Delimiters

Scanner defines where a token starts and ends based on a set of *delimiters*. The default delimiters are the whitespace characters, and this is the delimiter set that the preceding examples have used. However, it is possible to change the delimiters by calling the **useDelimiter( )** method, shown here:

Scanner useDelimiter(String *pattern*)
Scanner useDelimiter(Pattern *pattern*)

Here, *pattern* is a regular expression that specifies the delimiter set.

```
// Use Scanner to compute an average a list of comma-separated values.
import java.util.*;
import java.io.*;
class SetDelimiters
{       public static void main(String args[])  throws IOException
        {       int count = 0;
                double sum = 0.0;
                // Write output to a file.
                        FileWriter fout = new FileWriter("test.txt");
                // Now, store values in comma-separated list.
                        fout.write("2, 3.4, 5,6, 7.4, 9.1, 10.5, done");
                        fout.close();
                FileReader fin = new FileReader("Test.txt");
                Scanner src = new Scanner(fin);
                // Set delimiters to space and comma.
                        src.useDelimiter(", *");
                // Read and sum numbers.
                while(src.hasNext())
                {       if(src.hasNextDouble())
                        {       sum += src.nextDouble();
                                count++;
                        }
                        else
                        {       String str = src.next();
                                if(str.equals("done"))
                                        break;
                                else
                                {       System.out.println("File format error.");
                                        return;
                                }
                        }
                }
                src.close();
                System.out.println("Average is " + sum / count);
        }
}
```

In this version, the numbers written to **test.txt** are separated by commas and spaces. The use of the delimiter pattern **", * "** tells **Scanner** to match a comma and zero or more spaces as delimiters. The output is the same as before. You can obtain the current delimiter pattern by calling **delimiter( )**, shown here:

Pattern delimiter( )

Other Scanner Features

Scanner defines several other methods in addition to those already discussed. One that is particularly useful in some circumstances is **findInLine( )**. Its general forms are shown here:

String findInLine(Pattern *pattern*)
String findInLine(String *pattern*)

This method searches for the specified pattern within the next line of text. If the pattern is found, the matching token is consumed and returned. Otherwise, null is returned. It operates independently of any delimiter set. This method is useful if you want to locate a specific pattern. For example, the following program locates the Age field in the input string and then displays the age:

```
// Demonstrate findInLine().
import java.util.*;
class FindInLineDemo
{       public static void main(String args[])
        {       String instr = "Name: Tom Age: 28 ID: 77";
                // Find and display age.
                        conin.findInLine("Age:"); // find Age
                        if(conin.hasNext())
                                System.out.println(conin.next());
                        else
                                System.out.println("Error!");
                        conin.close();
        }
}
```

The output is **28**.

Related to **findInLine( )** is **findWithinHorizon( )**. It is shown here:

String findWithinHorizon(Pattern *pattern*, int *count*)
String findWithinHorizon(String *pattern*, int *count*)

This method attempts to find an occurrence of the specified pattern within the next *count* characters. If successful, it returns the matching pattern. Otherwise, it returns **null**. If *count* is zero, then all input is searched until either a match is found or the end of input is encountered.

You can bypass a pattern using **skip( )**, shown here:

```
Scanner skip(Pattern pattern)
Scanner skip(String pattern)
```

If *pattern* is matched, **skip( )** simply advances beyond it and returns a reference to the invoking object. If pattern is not found, **skip( )** throws **NoSuchElementException**. Other **Scanner** methods include **radix( )**, which returns the default radix used by the **Scanner**; **useRadix( )**, which sets the radix; **reset( )**, which resets the scanner; and **close( )**, which closes the scanner.