Java Collections Framework

- A *collection* is a container that represents a group of objects, often referred to as *elements*.

- Collections were added in 1.2 but went through modifications in 1.5 version (due to generics and for each style addition) and is present in java.util package.

- Prior to the Collections Framework, Java provided ad hoc classes such as **Dictionary, Vector, Stack, HashTable and Properties** to store and manipulate groups of objects.

- The Java Collections Framework supports three types of collections, named ***sets, lists,* and *maps*.**

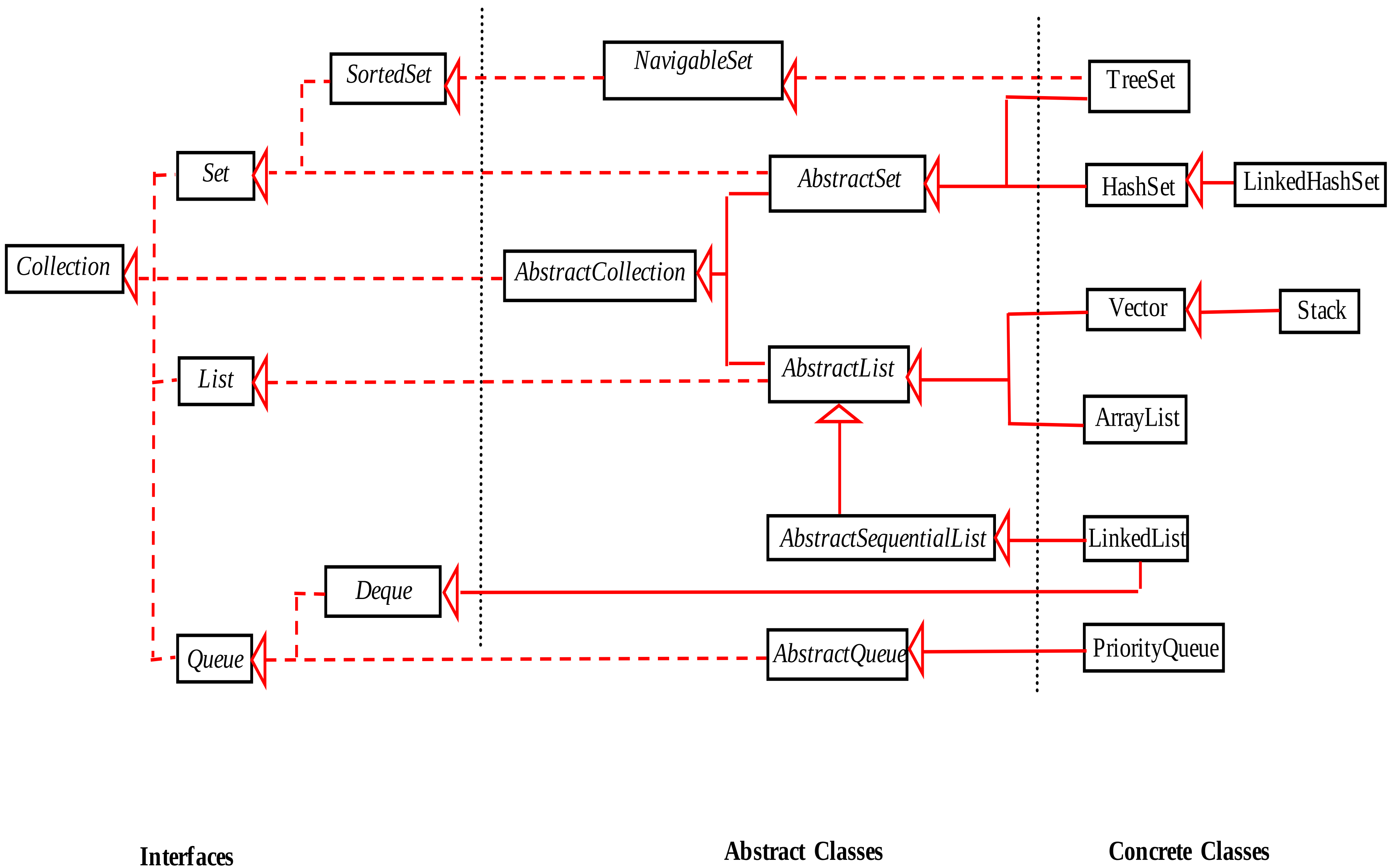
- The Collections Framework was designed to meet several goals-

1. **High-performance.-**The implementations for the fundamental collections (dynamic arrays , linked lists, trees, and hash tables) must be highly efficient.

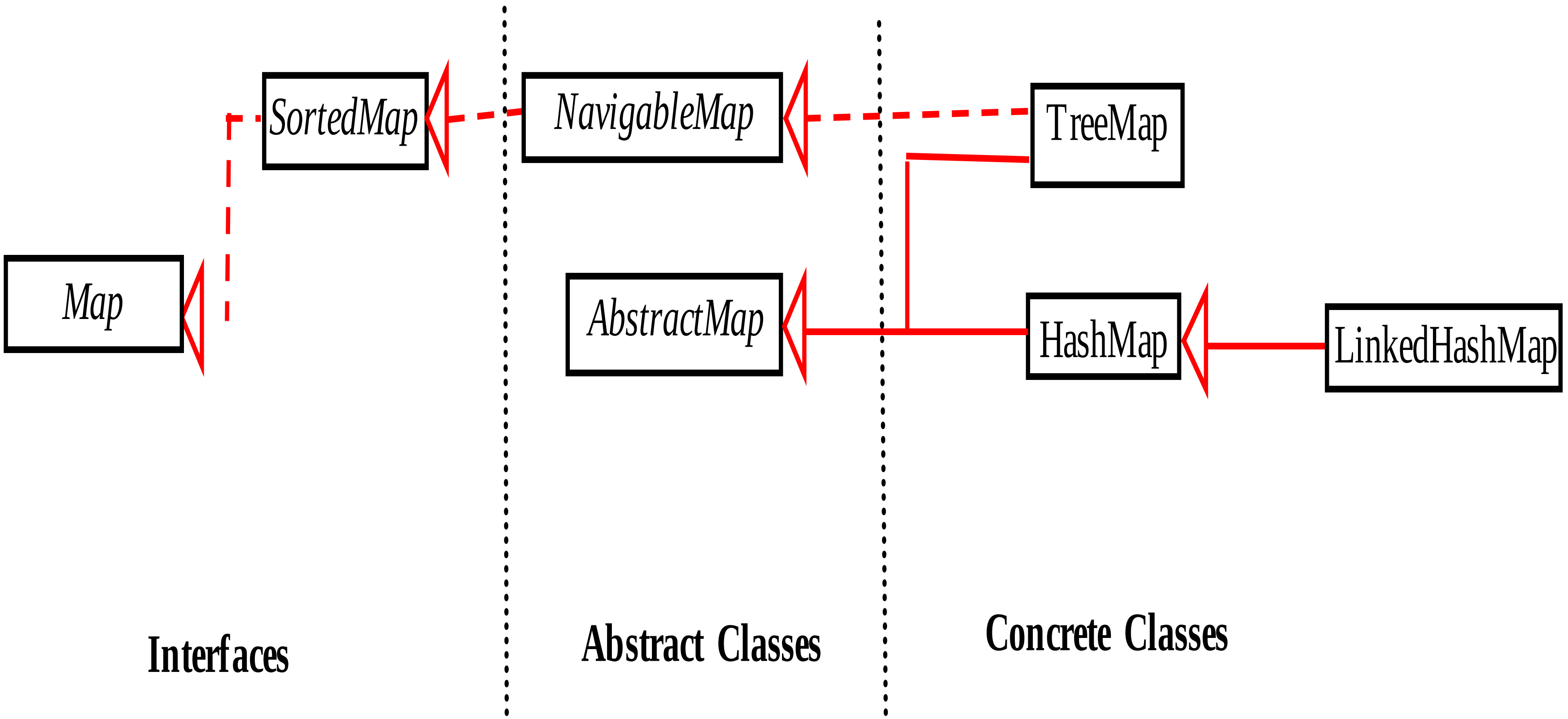
2. **Interoperability -** the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.

3. **Extending and/or adapting** a collection had to be easy. Hence, the entire Collections Framework is built upon a set of standard interfaces.

Javas 1D Collection Framework hierarchy



Maps (2D Collection)



The Collection Interfaces

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

Important methods of Collection Interface

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

The List Interface

- The **List interface extends Collection** and 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 -

  interface List<T>

Important methods of List Interface

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

Important methods of List Interface

ListIterator<E> listIterator( )	Returns an iterator to the start of the invoking list.
ListIterator<E> listIterator(int *index*)	Returns an iterator to the invoking list that begins at the specified index.
E remove(int *index*)	Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list.
List<E> subList(int *start*, int *end*)	Returns a list that includes elements from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

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

ArrayList

- The **ArrayList class extends AbstractList** and **implements List interface.**

- ArrayList is a generic class – **class ArrayList<T>**

- ArrayList supports dynamic arrays that can grow as needed.

- ArrayList has the following constructors :
  1. ArrayList( )
  2. ArrayList(Collection<? extends T> *c)*
  3. ArrayList(int *capacity)*

- One can increase the capacity of an **ArrayList** object manually by calling **ensureCapacity( )**

  - By increasing its capacity once, at the start, you can prevent several reallocations later.
  - Because reallocations are costly in terms of time, preventing unnecessary ones improves performance.

    void ensureCapacity(int *cap)*

- To reduce the size of the array that underlies an **ArrayList** object so that it is precisely as large as the number of items that it is currently holding, use **trimToSize( )**

  void trimToSize( )

```java
// Demonstrate ArrayList.
import java.util.*;

class ArrayListDemo {
  public static void main(String args[]) {
    // Create an array list.
    ArrayList<String> al = new ArrayList<String>();

    System.out.println("Initial size of al: " +
                       al.size());

    // Add elements to the array list.
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");
    al.add(1, "A2");

    System.out.println("Size of al after additions: " +
                       al.size());

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

    // Remove elements from the array list.
    al.remove("F");
    al.remove(2);

    System.out.println("Size of al after deletions: " +
                       al.size());
```

```
    System.out.println("Contents of al: " + al);
  }
}
```

The output from this program is shown here:

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

Methods Defined by Interface Queue

- **boolean offer(E obj):** Attempts to add obj to the queue. Returns **true** if obj was added and false otherwise.

- **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.

- **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 element( ):** Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty.

- **E peek(**) : Returns the element at the head of the queue. It returns **null** if the queue is empty. The element is not removed.

Methods Defined by Interface Deque

- **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.

- **boolean offerFirst(E obj):** Attempts to add obj to the head of the deque. Returns true if obj was added and 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 removeFirst(**): Returns the element at the head of the deque, removing the element in the process. It throws **NoSuchElementException** if deque is empty.
- **E removeLast( )** Returns the element at tail of the deque, removing element in the process. It throws **NoSuchElementException** if deque is empty.

Methods Defined by Interface Deque

- **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 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.

- **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.

LinkedList

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

- It provides a linked-list data structure.

- LinkedList is a generic class

  class LinkedList<T>

- LinkedList has the two constructors
  1. LinkedList( )
  2. LinkedList(Collection<? extends T> *c)*

LinkedList

Because **LinkedList implements the Deque** interface**,** many methods are available such as –

- **addFirst( ) or offerFirst( ) -** To add elements to the start of a list
- **addLast( ) or offerLast( ) -** To add elements to the end of the list.
- **getFirst( ) or peekFirst( ) -** To obtain the first element.
- **getLast( ) or peekLast( ) -** To obtain the last element.
- **removeFirst( ) or pollFirst( ) -** To remove the first element.
- **removeLast( ) or pollLast( ) -** To remove the last element.

```
// Demonstrate LinkedList.
import java.util.*;

class LinkedListDemo {
  public static void main(String args[]) {
    // Create a linked list.
    LinkedList<String> ll = new LinkedList<String>();

    // Add elements to the linked list.
    ll.add("F");
    ll.add("B");
    ll.add("D");
    ll.add("E");
    ll.add("C");
    ll.addLast("Z");
    ll.addFirst("A");

    ll.add(1, "A2");

    System.out.println("Original contents of ll: " + ll);

    // Remove elements from the linked list.
    ll.remove("F");
    ll.remove(2);
```

```
    System.out.println("Contents of ll after deletion: "
                       + ll);

    // Remove first and last elements.
    ll.removeFirst();
    ll.removeLast();

    System.out.println("ll after deleting first and last: "
                       + ll);

    // Get and set a value.

      String val = 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]
```

```java
// Stack Interface  MyStack.java

public interface MyStack<T>
{
        void push(T ob);
        T pop();
        int Size();
        T topElement();
         void display();
}
```

// ArrayList Implementation StackAL.java

```
import java.util.ArrayList;
public class StackAL<T> implements MyStack<T>
{   ArrayList<T> st;
    StackAL()
    {       st=new ArrayList<T>();
    }
    public void push(T ob)
    {       st.add(ob);
    }
    public T pop()
    {       T ob=null;
            if(!st.isEmpty())
            {     ob=st.remove(st.size()-1);
                   return ob;
            }
            return ob;
    }
```

```
public int stSize()
{       return st.size();
}
public T topEleMENT()
{       T ob=null;
        if(!st.isEmpty())
        {           ob=st.get(st.size()-1);
                    return ob;
        }
        return ob;
}
public void display()
{       System.out.println("Stack elements      using for each ");
        for(T se: st)
                    System.out.println(se);
        System.out.println("Stack elements      using object " + st);
}
}
```

// LinkedList Implementation StackLL.java

```
import java.util.LinkedList;
public class StackLL<T> implements MyStack<T>
{   LinkedList<T> st;
    StackLL()
    {       st=new LinkedList<T>();
    }
    public void push(T ob)
    {       st.addFirst(ob);
    }
    public T pop()
    {       T ob=null;
            if(!st.isEmpty())
            {           ob=st.removeFirst();
                        return ob;
            }
            return ob;
    }
```

```
    public int stSize()
    {   return st.size();
    }
    public T topElement()
    {   T ob=null;
        if(!st.isEmpty())
        {       ob=st.getFirst();
                return ob;
        }
        return ob;
    }
    public void display()
    {   System.out.println("Stack elements using for each ");
        for(T se: st)
                System.out.println(se);
        System.out.println("Stack elements using object " + st);
    }
}
```

// Stack Implementation Class StackImpl.java

```
import java.util.Scanner;
public class StackDemo
{     public static void main(String[] a) throws Exception
      {         MyStack<Integer>  stint;
                Scanner s=new Scanner(System.in);
                System.out.println(" Enter the Stack Class to be used ");
                String sclass=  s.next();
                Class<?> c=Class.forName(sclass);
                stint=(MyStack<Integer>)c.newInstance();

                stint.push(10);          stint.push(20);          stint.push(30);

                System.out.println(" Top element "+ stint.topelement());



                stint.pop();
                System.out.println(" pop element "+ stint.pop());


      }
}
```

The Set Interface

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

  interface Set<E>

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>

- *The result of hashing is called as hashcode.*
- Execution time of **add( ), contains( ), remove( ), and size( ) remain constant even for large sets.**

- The following constructors are defined:

  HashSet( ) // The default capacity is **16.**

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

  HashSet(int *capacity)*

  HashSet(int *capacity*, *float fillRatio)* *//Fill ratio is Load factor*

The HashSet Class

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

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

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

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

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

// Program to demonstrate HashSet.

```
import java.util.*;
class HashSetDemo {
public static void main(String args[]) {
// Create a hash set.
    HashSet<String> hs = new HashSet<String>();
// Add elements to the hash set.
      hs.add("B");
      hs.add("A");
      hs.add("D");
      hs.add("E");
      hs.add("C");
      hs.add("F");
      System.out.println(hs);
}
}
```

The following is the output from this program:

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

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

The LinkedHashSet Class

- The **LinkedHashSet class extends HashSet and adds no members of its own.**
- It is a generic class that has this declaration:

  class LinkedHashSet<E>

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

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

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

The LinkedHashSet Class

- To see the effect of **LinkedHashSet, s**ubstitute **LinkedHashSet for HashSet in the preceding program.**

- **The output will be**

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

- which is the order in which the elements were inserted.

The SortedSet Interface

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

interface SortedSet<E>

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

SortedSet Interface

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

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

The NavigableSet Interface

- The **NavigableSet** interface was added by **Java SE 6.**

- It inherits from **SortedSet, and adds some methods.**

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

- NavigableSet is a **generic interface** that has this declaration:

  interface NavigableSet<E>

The NavigableSet Interface

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

The NavigableSet Interface

NavigableSet<E> subSet(E *lowerBound*, boolean *lowIncl*, E *upperBound*, boolean *highIncl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
NavigableSet<E> tailSet(E *lowerBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound*. If *incl* is **true**, then an element equal to **lowerBound** is included. The resulting set is backed by the invoking set.

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

TreeSet Class

- **TreeSet** extends **AbstractSet** and implements the **NavigableSet** interface.
- It creates a collection that uses **a tree** for storage. Objects are stored in sorted, ascending order.

- Access and retrieval times are quite fast, which makes **TreeSet** an excellent choice when **storing large amounts of sorted information** that must be found quickly.
- TreeSet is a generic class that has this declaration:

  class TreeSet<E>

- **TreeSet has the following constructors:**
- TreeSet( ) // builds a tree in ascending order according to natural order
- TreeSet(Comparator<? super E> *cmp) // order specified by comparator*
- TreeSet(Collection<? extends E> *c)*
- TreeSet(SortedSet<E> *ss)*

Program to Demostrate TreeSet Class

```
import java.util.*;
class TreeSetDemo {
public static void main(String args[]) {
// Create a tree set.
TreeSet<String> ts = new TreeSet<String>();
// Add elements to the tree set.
ts.add("C");
ts.add("A");
ts.add("B");
ts.add("E");
ts.add("F");
ts.add("D");
System.out.println(ts);
} }
```

The output from this program is shown here:

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

Program to Demostrate TreeSet Class

```
import java.util.*;
class TreeSetDemo {
public static void main(String args[]) {
// Create a tree set.
TreeSet<String> ts = new TreeSet<String>();
// Add elements to the tree set.
ts.add("C");
ts.add("A");
ts.add("B");
ts.add("E");
ts.add("F");
ts.add("D");
System.out.println(ts);
} }
```

The output from this program is shown here:

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

Iterator

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

- **Iterator is a** generic interface which is declared as :

  interface Iterator<E>

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

- Thus, **Iterator enables us to cycle** through a collection, i.e it provides a means of *enumerating the contents of a collection*.

- Because each **Collection implements Iterator**, the elements of any collection class can be accessed through the methods defined by **Iterator**.

Iterator

Method	Description
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.
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( )**.

TABLE 17-8 The Methods Defined by **Iterator**

Using an Iterator

- Before we can access a collection through an iterator, we must obtain one.

- Each of the collection classes provides an **iterator( ) method that returns an iterator to the start of** the collection.

- In general, to use an iterator to cycle through the contents of a collection, follow these steps:
  1. Obtain an iterator to the start of the collection by calling the collection's **iterator( )** method.
  2. Set up a loop that makes a call to **hasNext( ). Have the loop iterate as long as hasNext( )** returns **true.**
  3. Within the loop, obtain each element by calling **next( ).**

List Iterator

- **ListIterator extends Iterator to allow** bidirectional traversal of a list, and the modification of elements.

- **ListIterator is a** generic interface which is declared as shown:

  interface ListIterator<E>

List Iterator

Method	**Description**
void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to **next( )**.
boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**.
boolean hasPrevious( )	Returns **true** if there is a previous element. Otherwise, returns **false**.
E next( )	Returns the next element. A **NoSuchElementException** is thrown if there is not a next element.
int nextIndex( )	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous( )	Returns the previous element. A **NoSuchElementException** is thrown if there is not a previous element.
int previousIndex( )	Returns the index of the previous element. If there is not a previous element, returns –1.
void remove( )	Removes the current element from the list. An **IllegalStateException** is thrown if **remove( )** is called before **next( )** or **previous( )** is invoked.
void set(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

TABLE 17-9 The Methods Defined by **ListIterator**

Using an Iterator

```
// Demonstrate iterators.
import java.util.*;
class IteratorDemo {
public static void main(String args[]) {
// Create an array list.
ArrayList<String>  al = new ArrayList<String>();

// Add elements to the array list.
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
```

Using an Iterator

```
// Use iterator to display contents of al.
System.out.print("Original contents of al using Iterator: ");
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 + "+");
}
```

Using an Iterator

```
System.out.print("Modified contents of al: " + al);
// Now, display the list backwards.
System.out.print("Modified list backwards: ");
while(litr.hasPrevious()) {
String element = litr.previous();
System.out.print(element + " ");
}
System.out.println();
}}
```

The output is shown here:

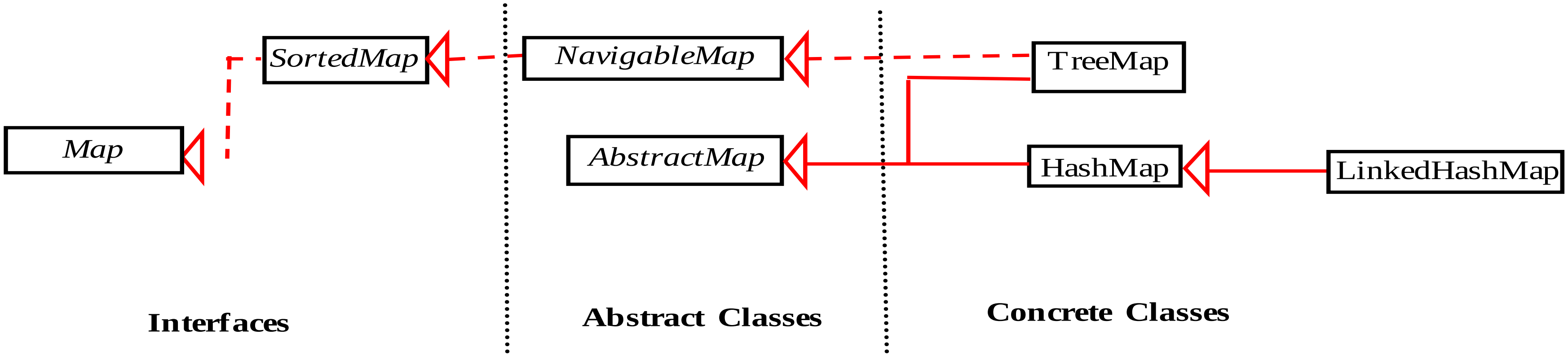
Original contents of al: C A E B D F

Modified contents of al: C+ A+ E+ B+ D+ F+

Modified list backwards: F+ D+ B+ E+ A+ C+

Maps

- A **map** is an object that stores **key/value** pairs.
- Given a key, we can find its value.
- Both keys and values are objects.
- **Keys** must be unique, but **Values** may be duplicated.
- We cannot cycle through a map using a for-each style **for loop.**



The Map Interface

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

- Map is generic :

  interface Map<K, V>

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

- We can obtain a collection-view of a map by using the **entrySet( )** method.
- entrySet( ) returns a Set containing the map entries.
- Each of these set elements is a **Map.Entry** object.

The Map.Entry Interface

- The Map.Entry interface enables us to work with a map entry.
- Map.Entry is generic inner interface in Map :

  interface Map.Entry<K, V>

The Map Interface

- 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.

- We can obtain a collection-view of a map by using 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( ).**
- **Collection-views** are the means by which maps are integrated into the larger Collections Framework.

Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
boolean containsKey(Object *k*)	Returns **true** if the invoking map contains *k* as a key. Otherwise, returns **false**.
boolean containsValue(Object *v*)	Returns **true** if the map contains *v* as a value. Otherwise, returns **false**.
Set<Map.Entry<K, V>> entrySet( )	Returns a **Set** that contains the entries in the map. The set contains objects of type **Map.Entry**. Thus, this method provides a set-view of the invoking map.
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map** and contains the same entries. Otherwise, returns **false**.
V get(Object *k*)	Returns the value associated with the key *k*. Returns **null** if the key is not found.
int hashCode( )	Returns the hash code for the invoking map.
boolean isEmpty( )	Returns **true** if the invoking map is empty. Otherwise, returns **false**.
Set<K> keySet( )	Returns a **Set** that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
V put(K *k*, V *v*)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are *k* and *v*, respectively. Returns **null** if the key did not already exist. Otherwise, the previous value linked to the key is returned.
void putAll(Map<? extends K, ? extends V> *m*)	Puts all the entries from *m* into this map.
V remove(Object *k*)	Removes the entry whose key equals *k*.
int size( )	Returns the number of key/value pairs in the map.
Collection<V> values( )	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

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

The Map.Entry Interface

- The **Map.Entry interface** enables us to work with a map entry.
- **entrySet( )** method declared by the Map interface **returns a Set containing the map entries.**
- Each of these set elements is a **Map.Entry object.**
- Map.Entry is generic and is declared like this:

  interface Map.Entry<K, V>

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

The Map.Entry Interface

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

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

The Map classes

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.

An entry in a **WeakHashMap** will automatically be removed when its key is no longer in ordinary use.

The HashMap Class

- The **HashMap** class extends AbstractMap and implements the **Map** interface.
- It uses a **hash table** to store the map. This allows the execution time of **get()** and **put( )** to remain constant even for large sets.
- **HashMap** is a generic class

class HashMap<K, V>

- HashMap **does not** add any methods of its own.

Constructors :

- HashMap( ) // default capacity is 16, default fill ratio is 0.75
- HashMap(Map<? extends K, ? extends V> *m)*
- HashMap(int *capacity)*
- HashMap(int *capacity*, *float fillRatio)*

- HashMap **does not guarantee the order** of its elements.

- Therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator

Program to illustrate HashMap. It maps names to account balances.

```
import java.util.*;
class HashMapDemo {
public static void main(String args[]) {
// Create a hash map.
HashMap<String, Double> hm = new HashMap<String, Double>();

// Put elements to the map
hm.put("John ", new Double(3434.34));
hm.put("Tom Smith", new Double(123.22));
hm.put("Jane Baker", new Double(1378.00));
hm.put("Tod Hall", new Double(99.22));
hm.put("Ralph Smith", new Double(-19.08));
```

Program to illustrate HashMap. It maps names to account balances.

```
// Get a set of the entries.
Set<Map.Entry<String, Double>> myset = hm.entrySet();
// Display the set.
for(Map.Entry<String, Double> me : myset) {
System.out.print(me.getKey() + ": ");
System.out.println(me.getValue());
}
System.out.println();
System.out.println(hm);
// Deposit 1000 into John 's account.
double balance = hm.get("John ");
hm.put("John ", balance + 1000);
System.out.println("John 's new balance: " + hm.get("John "));
}
}
```

LinkedHashMap

- **LinkedHashMap extends HashMap.**
- It maintains a linked list of the entries in the map, in the order in which they **were inserted.**
- We 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>
- **LinkedHashMap defines the following constructors:**
- LinkedHashMap( )
- LinkedHashMap(Map<? extends K, ? extends V> *m)*
- LinkedHashMap(int *capacity)*
- LinkedHashMap(int *capacity, float fillRatio)*
- LinkedHashMap(int *capacity, float fillRatio, boolean Order)*

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

LinkedHashMap

- 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 we override this method and return **true**, we can have **LinkedHashMap remove the oldest entry in the map.**

- **To keep the oldest entry, return false.**

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>
- Sorted maps allow very efficient manipulations of *submaps (in other words, subsets of a* map).
- To obtain a submap, use **headMap( ), tailMap( ), or subMap( ).**
- To get the first key in the set, call **firstKey( ).**
- To get the last key, **use lastKey( ).**

SortedMap Interface

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

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

NavigableMap Interface

- The **NavigableMap interface was added by Java SE 6.**

- It 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>

NavigableMap Interface

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

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

NavigableMap Interface

Method	Description
K higherKey(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
Map.Entry<K,V> lastEntry( )	Returns the last entry in the map. This is the entry with the largest key.
Map.Entry<K,V> lowerEntry(K *obj*)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K lowerKey(K *obj*)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableSet<K> navigableKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map. The resulting set is backed by the invoking map.
Map.Entry<K,V> pollFirstEntry( )	Returns the first entry, removing the entry in the process. Because the map is sorted, this is the entry with the least key value. **null** is returned if the map is empty.
Map.Entry<K,V> pollLastEntry( )	Returns the last entry, removing the entry in the process. Because the map is sorted, this is the entry with the greatest key value. **null** is returned if the map is empty.
NavigableMap<K,V> 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.

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

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.
- A tree map guarantees that its elements will **be sorted in ascending key order.**
- TreeMap is a generic class that has this declaration:

  class TreeMap<K, V>
- It adds no methods of its own.
- The following **TreeMap constructors are defined:**

  TreeMap( )

  TreeMap(Comparator<? super K> *comp)*

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

  TreeMap(SortedMap<K, ? extends V> *sm)*

TreeMap Class

```
import java.util.*;
class TreeMapDemo {
public static void main(String args[]) {

// Create a tree map.
TreeMap<String, Double> tm = new TreeMap<String, Double>();

// 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();
```

TreeMap Class

```
// 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:

Jane Baker: 1378.0

John Doe: 3434.34

Ralph Smith: -19.08

Todd Hall: 99.22

Tom Smith: 123.22 John Doe’s current balance: 4434.34