The Collection Classes

The ArrayList Class

The **ArrayList** class extends **AbstractList** and implements the **List** interface. **ArrayList** is ageneric class that has this declaration:

class ArrayList<E>

ArrayList has the constructors shown here:

ArrayList( )
ArrayList(Collection<? extends E> *c*)ArrayList(int *capacity*)

The first constructor builds an empty array list. The second constructor builds an array list that is initialized with the elements of the collection *c*. The third constructor builds an arraylist that has the specified initial *capacity*. The capacity is the size of the underlying array thatis used to store the elements. The capacity grows automatically as elements are added to anarray list.

```
// 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);
}}
```

void ensureCapacity(int *cap*)Here, *cap* is the new capacity.

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

Whatever the reason, converting an **ArrayList** to an array is a trivial matter.

The LinkedList Class

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

class LinkedList<E>
Here, **E** specifies the type of objects that the list will hold. **LinkedList** has the two constructorsshown here:

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

The first constructor builds an empty linked list. The second constructor builds a linked listthat is initialized with the elements of the collection *c*.
The following program illustrates **LinkedList**:

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

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

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

ll.add(1, "A2");

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

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

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

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

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

// Get and set a value.
String val = ll.get(2); ll.set(2, val
+ " Changed");

System.out.println("ll after change: " + ll);
}
}
```

Because **LinkedList** implements the **List** interface, calls to **add(E)** append items to the endof the list, as do calls to **addLast( )**. To insert items at a specific location, use the **add(int, E)** form of **add( )**, as illustrated by the call to **add(1, "A2")** in the example.

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.

As most readers likely know, a hash table stores information by using a mechanism calledhashing. In *hashing,* the informational content of a key is used to determine a unique value, called its *hash code.* 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.
The following constructors are defined:

HashSet( )
HashSet(Collection<? extends E> *c*)HashSet(int
capacity)
HashSet(int *capacity*, float *fillRatio*)

HashSet does not define any additional methods beyond those provided by its superclassesand interfaces.
Here is an example that demonstrates **HashSet**:

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

class HashSetDemo {
public static void main(String args[]) {
```

```
// Create a hash set.
HashSet<String> hs = new HashSet<String>();

// Add elements to the hash set.
hs.add("B");
hs.add("A");
hs.add("D");
hs.add("E");
hs.add("C");
hs.add("F");

System.out.println(hs);
}
}
```

The LinkedHashSet Class

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

class LinkedHashSet<E>

Here, **E** specifies the type of objects that the set will hold. Its constructors parallel those in **HashSet**.

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.

The 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 largeamounts of sorted information that must be found quickly.
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( )
TreeSet(Collection<? extends E> *c*)
TreeSet(Comparator<? super E> *comp*)
TreeSet(SortedSet<E> *ss*)

The first form constructs an empty tree set that will be sorted in ascending order accordingto the natural order of its elements. The second form builds a tree set that contains the elementsof *c*. The third form constructs an empty tree set that will be sorted according to the comparatorspecified by *comp.*

The PriorityQueue Class

PriorityQueue extends **AbstractQueue** and implements the **Queue** interface. It creates a queuethat is prioritized based on the queue's comparator. **PriorityQueue** is a generic class that has this declaration:

class PriorityQueue<E>

Here, **E** specifies the type of objects stored in the queue. **PriorityQueue**s are dynamic, growing as necessary.
PriorityQueue defines the six constructors shown here:

PriorityQueue( ) PriorityQueue(int *capacity*)
PriorityQueue(int *capacity*, Comparator<? super E> *comp*)
PriorityQueue(Collection<? extends E> *c*)
PriorityQueue(PriorityQueue<? extends E> *c*)
PriorityQueue(SortedSet<? extends E> *c*)

The first constructor builds an empty queue. Its starting capacity is 11. The second constructor builds a queue that has the specified initial capacity. The third constructor builds a queue with the specified capacity and comparator. The last three constructors create queues that are initialized with the elements of the collection passed in *c.* In all cases, the capacity grows automatically as elements are added.

The ArrayDeque Class

Java SE 6 added the **ArrayDeque** class, which extends **AbstractCollection** and implements the **Deque** interface. It adds no methods of its own. **ArrayDeque** creates a dynamic array and has no capacity restrictions. (The **Deque** interface supports implementations that restrict capacity, but does not require such restrictions.)
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( ) ArrayDeque(int *size*)

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

The first constructor builds an empty deque. Its starting capacity is 16. The second constructor builds a deque that has the specified initial capacity. The third constructor creates a deque that is initialized with the elements of the collection passed in *c*. In all cases, the capacity grows as needed to handle the elements added to the deque.

The EnumSet Class

EnumSet extends **AbstractSet** and implements **Set**. It is specifically for use with keys of an **enum** type. It is a generic class that has this declaration:

class EnumSet<E extends Enum<E>>

Here, **E** specifies the elements. Notice that **E** must extend **Enum<E>**, which enforces the requirement that the elements must be of the specified **enum** type.

EnumSet defines no constructors. Instead, it uses the factory methods shown in Table 17-7to create objects.

Accessing a Collection via an Iterator

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element. One way to do this is to employ an *iterator,* which is an objectthat implements either the **Iterator** or the **ListIterator** interface. **Iterator** enables you to cyclethrough a collection, obtaining or removing elements. **ListIterator** extends **Iterator** to allow

Method	**Description**
static <E extends Enum<E>> EnumSet<E> allOf(Class<E> *t*)	Creates an **EnumSet** that contains the elements in theenumeration specified by *t*.
static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> *e*)	Creates an **EnumSet** that is comprised of those elements notstored in *e*.
static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> *c*)	Creates an **EnumSet** from the elements stored in *c*.
static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> *c*)	Creates an **EnumSet** from the elements stored in *c*.
static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> *t*)	Creates an **EnumSet** that contains the elements that are not inthe enumeration specified by *t*, which is an empty set by definition.
static <E extends Enum<E>> EnumSet<E> of(E *v*, E ... *varargs*)	Creates an **EnumSet** that contains *v* and zero or moreadditional enumeration values.
static <E extends Enum<E>> EnumSet<E> of(E *v*)	Creates an **EnumSet** that contains *v*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*)	Creates an **EnumSet** that contains *v1* and *v2*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*)	Creates an **EnumSet** that contains *v1* through *v3*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*)	Creates an **EnumSet** that contains *v1* through *v4*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*, E *v5*)	Creates an **EnumSet** that contains *v1* through *v5*.
static <E extends Enum<E>> EnumSet<E> range(E *start*, E *end*)	Creates an **EnumSet** that contains the elements in the rangespecified by *start* and *end*.

TABLE 17-7 The Methods Defined by **EnumSet**

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 nota next element.
void remove( )	Removes the current element. Throws **IllegalStateException** if an attemptis made to call **remove( )** that is not preceded by a call to **next( )**.

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

bidirectional traversal of a list, and the modification of elements. **Iterator** and **ListIterator** are generic interfaces which are declared as shown here:

```
interface Iterator<E>
interface ListIterator<E>
```

Here, **E** specifies the type of objects being iterated. The **Iterator** interface declares the methodsshown in Table 17-8. The methods declared by **ListIterator.**

Using an Iterator

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an **iterator( )** method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one

Method	**Description**
void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returnedby 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 thrownif 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 thrownif there is not a previous element.
int previousIndex( )	Returns the index of the previous element. If there is not a previouselement, 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 returnedby a call to either **next( )** or **previous( )**.

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

element at a time. In general, to use an iterator to cycle through the contents of a collection,follow these steps:

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

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

The For-Each Alternative to Iterators

If you won't be modifying the contents of a collection or obtaining elements in reverse order, then the for-each version of the **for** loop is often a more convenient alternative to cycling through a collection than is using an iterator. Recall that the **for** can cycle through any collectionof objects that implement the **Iterable** interface. Because all of the collection classes implementthis interface, they can all be operated upon by the **for**.

The following example uses a **for** loop to sum the contents of a collection:

```
// Use the for-each for loop to cycle through a
collection.import java.util.*;

class ForEachDemo {
  public static void main(String args[]) {
    // Create an array list for integers.
    ArrayList<Integer> vals = new
    ArrayList<Integer>();

    // Add values to the array
    list.vals.add(1);
    vals.add(2);
    vals.add(3);
    vals.add(4);
    vals.add(5);

    // Use for loop to display the values.
    System.out.print("Original 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);
        }
      }
```

Storing User-Defined Classes in Collections

For the sake of simplicity, the foregoing examples have stored built-in objects, such as **String** or **Integer**, in a collection. Of course, collections are not limited to the storage of built-in objects. Quite the contrary. The power of collections is that they can store any type of object,including objects of classes that you create.

```
      import java.util.*;
    class Address { private
      String name;
      private String street;
      private String city; private
      String state; private String
      code;

ddress(String n, String s, String c,String st,
      String cd) {
      name = n; street =
      s;city = c; state =
      st;code = cd;
      }

      public String toString() {
return name + "\n" + street + "\n" + city + " " +
      state + " " + code;
      }
      }

      class MailList {
    public static void main(String args[]) {
      LinkedList<Address> ml = new LinkedList<Address>();

      // Add elements to the linked list. ml.add(new
      Address("J.W. West", "11 Oak Ave",
      "Urbana", "IL", "61801"));
      ml.add(new Address("Ralph Baker", "1142 Maple Lane",
      "Mahomet", "IL", "61853"));
      ml.add(new Address("Tom Carlton", "867 Elm St",
      "Champaign", "IL", "61820"));

      // Display the mailing list.
      for(Address element : ml)
      System.out.println(element + "\n");
System.out.println();
}
}
```

The RandomAccess Interface

The **RandomAccess** interface contains no members. However, by implementing this interface,a collection signals that it supports efficient random access to its elements. Although a collection might support random access, it might not do so efficiently. By checking for the **RandomAccess**interface, client code can determine at run time whether a collection is suitable for certain types of random access operations—especially as they apply to large collections. (You can use **instanceof** to determine if a class implements an interface.) **RandomAccess** is implementedby **ArrayList** and by the legacy **Vector** class, among others.

Working with Maps

A *map* is an object that stores associations between keys and values, or *key/value pairs.* Givena key, you can find its value. Both keys and values are objects. The keys must be unique, butthe values may be duplicated. Some maps can accept a **null** key and **null** values, others cannot.

The Map Interfaces

Because the map interfaces define the character and nature of maps, this discussion of mapsbegins with them. 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**.
NavigableMa p	Extends **SortedMap** to handle the retrieval of entries based on closest-matchsearches. (Added by Java SE 6.)
SortedMap	Extends **Map** so that the keys are maintained in ascending order.

Each interface is examined next, in turn.

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. Afterthe value is stored, you can retrieve it by using its key. **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.

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 theinvoking 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 isnot 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 methodprovides 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 previousvalue 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 methodprovides a collection-view of the values in the map.

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

use **keySet( )**. To get a collection-view of the values, use **values( )**. Collection-views are themeans by which maps are integrated into the larger Collections Framework.

The SortedMap Interface

The **SortedMap** interface extends **Map**. It ensures that the entries are maintained in ascendingorder 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.

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

Method	Description
Comparator<? super K> comparator( )	Returns the invoking sorted map's comparator. If naturalordering 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 areless 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 aregreater than or equal to *start* and less than *end*.
SortedMap<K, V> tailMap(K *start*)	Returns a map containing those entries with keys that aregreater than or equal to *start*.

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

The NavigableMap Interface

The **NavigableMap** interface was added by Java SE 6. It extends **SortedMap** and declaresthe behavior of a map that supports the retrieval of entries based on the closest match to agiven 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 withthe keys. In addition to the methods that it inherits from **SortedMap**, **NavigableMap** adds those summarized in Table 17-12.

Method	Description
Map.Entry<K,V> ceilingEntry(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a keyis 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 keyis found, it is returned. Otherwise, **null** is returned.
NavigableSet<K> descendingKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map inreverse 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. Theresulting 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 keyis 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 keyis 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 bythe 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 isfound, its entry is returned. Otherwise, **null** is returned.

The Map.Entry Interface

The **Map.Entry** interface enables you to work with a map entry. Recall that the **entrySet( )** method declared by the **Map** interface returns a **Set** containing the map entries. Each of theseset 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. Table 17-13 summarizesthe methods declared by **Map.Entry**. Various exceptions are possible.

Method	Description
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map.Entry** whose key and value are equal to that of theinvoking 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 notthe correct type for the map. An **IllegalArgumentException** is thrown if there isa 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.

The Map Classes

Several classes provide implementations of the map interfaces. The classes that can be usedfor 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.

Notice that **AbstractMap** is a superclass for all concrete map implementations.

WeakHashMap implements a map that uses “weak keys,” which allows an element ina map to be garbage-collected when its key is otherwise unused. This class is not discussedfurther here. The other map classes are described next.

The HashMap Class

The **HashMap** class extends **AbstractMap** and implements the **Map** interface. It uses a hashtable to store the map. This allows the execution time of **get( )** and **put( )** to remain constanteven for large sets. **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( )
HashMap(Map<? extends K, ? extends V> *m*)
HashMap(int *capacity*)
HashMap(int *capacity*, float *fillRatio*)

```
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 Doe", 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));

    // 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"));
  }
}
```

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 storingkey/value pairs in sorted order and allows rapid retrieval. You should note that, unlike a hash map, 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>

Here, **K** specifies the type of keys, and **V** specifies the type of values.
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 has no methods beyond those specified by the **NavigableMap** interface andthe **AbstractMap** class.
The following program reworks the preceding example so that it uses **TreeMap**:

```
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();

   // 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 LinkedHashMap Class

LinkedHashMap extends **HashMap**. It maintains a linked list of the entries in the map, in theorder 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 returnedin 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( )
LinkedHashMap(Map<? extends K, ? extends V> *m*)
LinkedHashMap(int *capacity*)
LinkedHashMap(int *capacity*, float *fillRatio*)
LinkedHashMap(int *capacity*, float *fillRatio*, boolean *Order*)

The first form constructs a default **LinkedHashMap**. The second form initializes the **LinkedHashMap** with the elements from *m.* The third form initializes the capacity. The fourthform initializes both capacity and fill ratio. The meaning of capacity and fill ratio are the sameas for **HashMap**. The default capactiy is 16. The default ratio is 0.75. The last form allows you to specify whether the elements will be stored in the linked list by insertion order, or byorder of last access. If *Order* is **true**, then access order is used. If *Order* is **false**, then insertionorder is used.

L**inkedHashMap** 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, thismethod returns **false** and does nothing. However, if you override this method, then you canhave the **LinkedHashMap** remove the oldest entry in the map. To do this, have your overridereturn **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. **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 API documentationexplicitly states that **IdentityHashMap** is not for general use.

The EnumMap Class

EnumMap extends **AbstractMap** and implements **Map**. It is specifically for use with keys ofan **enum** type. 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. Notice that **K** must extend **Enum<K>**, which enforces the requirement that the keys must be of an **enum** type.

EnumMap defines the following constructors:

EnumMap(Class<K> *kType*)
EnumMap(Map<K, ? extends V> *m*)
EnumMap(EnumMap<K, ? extends V>
em)

The first constructor creates an empty **EnumMap** of type *kType.* The second creates an **EnumMap** map that contains the same entries as *m.* The third creates an **EnumMap** initializedwith the values in *em.*

EnumMap defines no methods of its own.

Comparators

Both **TreeSet** and **TreeMap** store elements in sorted order. However, it is the comparator that defines precisely what "sorted order" means. Doing so gives you theability to govern precisely how elements are stored within sorted collections and maps.

Comparator is a generic interface that has this declaration:

interface Comparator<T>

Here, **T** specifies the type of objects being compared.

The **Comparator** interface defines two methods: **compare( )** and **equals( )**. The **compare( )** method, shown here, compares two elements for order:

int compare(T *obj1*, T *obj2*)

obj1 and *obj2* are the objects to be compared. This method returns zero if the objects are equal.It returns a positive value if *obj1* is greater than *obj2.*

The **equals( )** method, shown here, tests whether an object equals the invoking comparator:

boolean equals(Object *obj*)

Here, *obj* is the object to be tested for equality. The method returns **true** if *obj* and the invokingobject are both **Comparator** objects and use the same ordering. Otherwise, it returns **false**. Overriding **equals( )** is unnecessary, and most simple comparators will not do so.

Using a Comparator

The following is an example that demonstrates the power of a custom comparator. It implements the **compare( )** method for strings that operates in reverse of normal. Thus, it causes a tree set to be stored in reverse order.

```
// Use a custom
comparator.import
java.util.*;

// A reverse comparator for strings.
```

```
class MyComp implements
  Comparator<String> {public int
  compare(String a, String b) {
    String aStr, bStr;

    aStr = a;
    bStr = b;

    // Reverse the comparison.
    return bStr.compareTo(aStr);
  }

  // No need to override equals.
}

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 stored in reverse order:

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

```
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();

    // 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"));
  }
}
```

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

John Doe's new balance: 4434.34
```

The comparator class **TComp** compares two strings that hold first and last names. It does so by first comparing last names. To do this, it finds the index of the last space in each stringand then compares the substrings of each element that begin at that point. In cases where last names are equivalent, the first names are then compared. This yields a tree map that is sorted by last name, and within last name by first name. You can see this because Ralph Smith comesbefore Tom Smith in the output.

The Collection Algorithms

The Collections Framework defines several algorithms that can be applied to collections andmaps. These algorithms are defined as static methods within the **Collections** class. They aresummarized in Table 17-14. As explained earlier, beginning with JDK 5 all of the algorithmshave been retrofitted for generics. Although the generic syntax might seem a bit intimidatingat first, the algorithms are as simple to use as they were before generics. It's just that now, they are type safe.

Method	**Description**
static <T> boolean addAll(Collection <? super T> *c*, T ... *elements*)	Inserts the elements specified by *elements* into thecollection 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*. (Added by JavaSE 6.)
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 negativevalue 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 negativevalue 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 willcause a **ClassCastException**.
static <T> List<T> emptyList( )	Returns an immutable, empty **List** object of theinferred type.
static <K, V> Map<K, V> emptyMap( )	Returns an immutable, empty **Map** object of theinferred type.
static <T> Set<T> emptySet( )	Returns an immutable, empty **Set** object of theinferred type.
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*.

Several of the methods can throw a **ClassCastException**, which occurs when an attemptis made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection. Other exceptions are possible, depending on the method.

One thing to pay special attention to is the set of **checked** methods, such as **checkedCollection( )**, which returns what the API documentation refers to as a "dynamically typesafe view" of a collection. This view is a reference to the collection that monitors insertionsinto the collection for type compatibility at run time. An attempt to insert an incompatibleelement will cause a **ClassCastException**. Using such a view is especially helpful during debugging because it ensures that the collection always contains valid elements. Related methods include **checkedSet( )**, **checkedList( )**, **checkedMap( )**, and so on. They obtain a type-safe view for the indicated collection.

```
// 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
```

Notice that **min( )** and **max( )** operate on the list after it has been shuffled. Neither requiresa sorted list for its operation.

Why Generic Collections?

As mentioned at the start of this chapter, the entire Collections Framework was refitted forgenerics when JDK 5 was released. Furthermore, the Collections Framework is arguably the single most important use of generics in the Java API. The reason for this is that genericsadd type safety to the Collections Framework. Before moving on, it is worth taking some time to examine in detail the significance of this improvement.

Let's begin with an example that uses pre-generics code. The following program storesa list of strings in an **ArrayList** and then displays the contents of the list:

```
// Pre-generics example that uses a
collection.import java.util.*;

class OldStyle {
  public static void main(String
    args[]) {ArrayList list = new
    ArrayList();

    // These lines store strings, but any type of object
    // can be stored.  In old-style code, there is no
    // convenient way to restrict the type of objects stored
    // in a collection
    list.add("one");
    list.add("two");
    list.add("three");
    list.add("four");

    Iterator itr =
    list.iterator();
    while(itr.hasNext()) {

      // To retrieve an element, an explicit type cast is needed
      // because the collection stores only Object.
      String str = (String) itr.next(); // explicit cast needed here.

      System.out.println(str + " is " + str.length() + " chars long.");
    }
  }
}
```

The Legacy Classes and Interfaces

As explained at the start of this chapter, early versions of **java.util** did not include the Collections Framework. Instead, it defined several classes and an interface that provided anad 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 fully compatible with the framework. While no classes have actually been deprecated, one has beenrendered obsolete. Of course, where a collection duplicates the functionality of a legacy class, you will usually want to use the collection for new code. In general, the legacy classes are supported because there is still code that uses them.

One other point: none of the collection classes are synchronized, but all the legacy classes are synchronized. This distinction may be important in some situations. Of course, you caneasily synchronize collections, too, by using one of the algorithms provided by **Collections**.

The legacy classes defined by **java.util** are shown here:

Dictionary	Hashtable	Properties	Stack	Vector

There is one legacy interface called **Enumeration**. The following sections examine **Enumeration** and each of the legacy classes, in turn.

The Enumeration Interface

The **Enumeration** interface defines the methods by which you can *enumerate* (obtain one at a time) the elements in a collection of objects. This legacy interface has been superseded by **Iterator**. Although not deprecated, **Enumeration** is considered obsolete for new code. However,it is used by several methods defined by the legacy classes (such as **Vector** and **Properties**),is used by several other API classes, and is currently in widespread use in application code.Because it is still in use, it was retrofitted for generics by JDK 5. It has this declaration:

```
interface Enumeration<E>
```

where **E** specifies the type of element being enumerated.

Enumeration specifies the following two methods:

```
boolean hasMoreElements(
)E nextElement( )
```

When implemented, **hasMoreElements( )** must return **true** while there are still more elementsto extract, and **false** when all the elements have been enumerated. **nextElement( )** returns the next object in the enumeration. That is, each call to **nextElement( )** obtains the next object inthe enumeration. It throws **NoSuchElementException** when the enumeration is complete.

Vector

Vector implements a dynamic array. It is similar to **ArrayList**, but with two differences: **Vector** is synchronized, and it contains many legacy methods that are not part of the Collections Framework. With the advent of collections, **Vector** was reengineered to extend **AbstractList** andto implement the **List** interface. With the release of JDK 5, it was retrofitted for generics andreengineered to implement **Iterable**. This means that **Vector** is fully compatible with collections,and a **Vector** can have its contents iterated by the enhanced **for** loop.

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( )
```

Vector(int *size*)
Vector(int *size*, int *incr*)
Vector(Collection<? extends E> *c*)

The first form creates a default vector, which has an initial size of 10. The second form createsa vector whose initial capacity is specified by *size.* The third form 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. The fourth form creates a vector that contains the elements of collection *c.*

Vector defines these protected data members:

int capacityIncrement;
int elementCount;
Object[ ] elementData;

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 arrayspecified 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 thevector, –1 is returned.
int indexOf(Object *element*, int *start*)	Returns the index of the first occurrence of *element* at or after *start***.** If the objectis 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 moreelements.
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 thevector, –1 is returned.
int lastIndexOf(Object *element*, int *start*)	Returns the index of the last occurrence of *element* before *start***.** If the objectis 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 specifiedobject 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***.**

Stack

Stack is a subclass of **Vector** that implements a standard last-in, first-out stack. **Stack** onlydefines 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 17-16.

To put an object on the top of the stack, call **push( )**. To remove and return the top element, call **pop( )**. An **EmptyStackException** is thrown if you call **pop( )** when the invoking stack is empty. You can use **peek( )** to return, but not remove, the top object. The **empty( )** method returns **true** if nothing is on the stack

Method	Description
boolean empty( )	Returns **true** if the stack is empty, and returns **false** if the stackcontains 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 theprocess.
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 topof the stack is returned. Otherwise, –1 is returned.

Dictionary

Dictionary is an abstract class that represents a key/value storage repository and operates much like **Map**. Given a key and value, you can store the value in a **Dictionary** object. Oncethe value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can bethought of as a list of key/value pairs. Although not currently deprecated, **Dictionary** is classified as obsolete, because it is fully superseded by **Map**. However, **Dictionary** is still inuse and thus is fully discussed here.

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 methodsdefined by **Dictionary** are listed in Table 17-17.

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 itcontains 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

Hashtable was part of the original **java.util** and is a concrete implementation of a **Dictionary**.However, with the advent of collections, **Hashtable** was reengineered to also implement the**Map** interface. Thus, **Hashtable** is now integrated into the Collections Framework. It is similar to **HashMap**, but is synchronized.

Like **HashMap**, **Hashtable** stores key/value pairs in a hash table. However, neither keys nor values can be **null**. When using a **Hashtable**, you specify an object that is used as a key,and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table. **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.

A hash table can only store objects that override the **hashCode( )** and **equals( )** methods that are defined by **Object**. The **hashCode( )** method must compute and return the hash code for the object. Of course, **equals( )** compares two objects. Fortunately, many of Java's built-inclasses already implement the **hashCode( )** method. For example, the most common type of **Hashtable** uses a **String** object as the key. **String** implements both **hashCode( )** and **equals( )**.
The **Hashtable** constructors are shown here:

Hashtable( ) Hashtable(int *size*)
Hashtable(int *size*, float *fillRatio*) Hashtable(Map<?
extends K, ? extends V> *m*)