Chapter 13

Container Classes

As mentioned in Chapter 7, one-to-many and many-to-many associations between objects are common in object-oriented programming. These types of associations are best implemented using classes that allow groups or collections of objects to be stored. Object-oriented programming languages typically provide a number of classes that can be used for storing collections of objects. For example, the Java *Collections* framework provides several containers for storing objects in linked lists, trees, hash tables, and other structures. These containers are referred to as *collection classes*.

Two important issues with collection classes is generalizing the type of objects that can be stored and providing type safety. The chapter starts off by showing how the *generics* feature in Java addresses these two issues in an elegant way. This is followed by a discussion of the Iterator Design Pattern which prescribes a way for the elements of different types of containers to be traversed in a uniform manner. Next, the chapter describes the classes in the Java *Collections* framework. Finally, the chapter concludes with some guidelines for choosing the right collection class to implement an association between two classes.

The chapter gives a good overview of the Java *Collections* framework since it provides an excellent example of how fundamental object-oriented concepts such as interfaces, abstract classes, and inheritance can be applied to build a set of general-purpose classes. To fully appreciate the material discussed in this chapter, it will be helpful if you have a good background in data structures such as linked lists, binary trees, and hash tables. However, the material is presented from an object-oriented perspective and the internal details of the data structures are mostly ignored.

13.1 The Need for Generics

The Java *Collections* framework uses a feature known as *generics* which allows a container class to be specified in such a way that it is possible to store any type of object in a container and yet be type safe. This section explains why generics are important by comparing linked lists that can store different types of data. It also shows how a linked list can be implemented using generics.

13.1.1 Linked Lists

A *linked list* is a data structure consisting of a set of nodes in a sequence where each node is linked to the next node in the sequence. A node has two components, `element` and `next`. The `element` component stores data associated with the node. The `next` component stores a reference to the next node in the sequence. The `next` reference of the last node in the sequence is `null`. A variable `head` is used to keep track of the first node in the list. From `head`, it is possible to find all the nodes in the linked list by following the links in the `next` component of each node. Figure 13.1 is a diagram of a linked list which stores integer values.

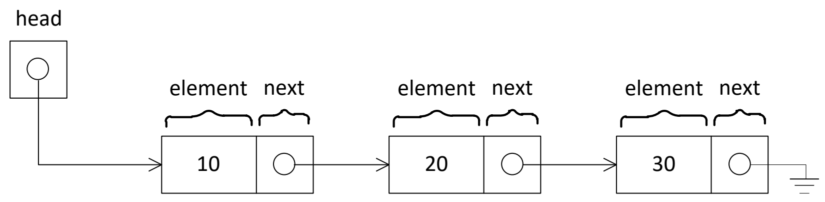

Figure 13.1: A Linked List

13.1.2 A Linked List of Integers

To implement the linked list shown in Figure 13.1, an inner class `Node` can be used to store the information on each node. `Node` is an inner class since it is highly unlikely that it will be used by other classes. The `LinkedList` class has two instance variables: `head`, which keeps track of the first node in the list, and `count`, which keeps track of the amount of nodes in the list. A portion of the code for the `LinkedList` class is as follows:

```
public class LinkedList {

   private Node head; // first node of the linked list
   private int count; // amount of nodes in the linked list
   public LinkedList() {
      head = null;
      count = 0;
   }

   public void addFirst(int element) {
      Node newNode = new Node(element);
      newNode.next = head;
      head = newNode;
      count++;
   }

   public int getFirst() {
      if (count > 0)
         return head.element;
      else
         throw new LinkedListException("List is empty.");
   }

   // other linked list methods

   // Node inner class

   private class Node {
      private int element;
      private Node next;

      private Node(int element) {
         this.element = element;
      }
   }

}
```

The important thing to observe in the code above is the declaration of the instance variable `element` as an `int` in the `Node` inner class:

```
private int element;
```

Since `element` is explicitly declared to be of type `int`, it is impossible to use the linked list to store values of other types such as `double`, `char`, `boolean`, etc. Thus, if a linked list is required to store other types of data, a new `LinkedList` class must be written where the `element` type in `Node` is changed to the type required. In addition, all the linked list methods that refer to `element` (e.g., `addFirst()`) must be also be changed to accommodate the new type. For these reasons, the `LinkedList` class above is not a very practical container.

13.1.3 A Linked List of Objects

An improvement on the linked list is to declare the `element` type as `Object`. Wherever `element` is used in the class definition, its type is changed to `Object`. The code for the revised version of `LinkedList` is given below.

```
public class LinkedList {

   private Node head; // first node of the linked list
   private int count; // amount of nodes in the linked list

   public LinkedList() {
      head = null;
      count = 0;
   }

   public void addFirst(Object element) {
      Node newNode = new Node(element);
      newNode.next = head;
      head = newNode;
      count++;
   }

   public Object getFirst() {
      if (count > 0)
         return head.element;
      else
         throw new LinkedListException("List is empty.");
   }



   // other linked list methods

   :
```

```
   // Node inner class

   private class Node {
      private Object element;
      private Node next;
                                              } Node inner class
      private Node(Object element) {
         this.element = element;
      }
   }

}
```

The `element` type of `Node` is declared to be of type `Object` as follows:

```
private Object element;
```

Since the `Object` class is the superclass of all classes in Java, by substitutability this means that instances of any class can be stored in the linked list.

Suppose `linkedList` is an instance of the revised version of the `LinkedList` class. Also, suppose that `account` is an instance of the `Account` class. We can insert `account` at the top of the linked list as follows:

```
linkedList.addFirst(account);
```

Now, suppose that `student` is an instance of the `Student` class. We can insert `student` in the linked list as follows:

```
linkedList.addFirst(student);
```

This reveals one important weakness when `element` is declared to be of type `Object`: it is not possible to prevent objects of different types from being stored in the linked list at the same time. This is because an instance of any class in Java is of type `Object`, by substitutability.

Using `Object` as the `element` type presents another problem when dealing with the reverse polymorphism problem. Suppose we legitimately store a set of objects of different types in the linked list (e.g., `Account`, `Student`). Any retrieval method (e.g., `getFirst()`) will return an object of type `Object` despite what it really is at run-time.

For example, consider the following code:

```
Object o = linkedList.getFirst();
```

Although `o` can either be an `Account` instance or a `Student` instance, the most we know about `o` after the call to `getFirst()` is that it is of type `Object`. To obtain the object actually stored in the container, the `instanceof` operator must be used to determine if the object is an `Account` or a `Student`; the object must then be cast to its specific type before any class-specific processing takes place:

```
if (o instanceof Account) {
   Account a = (Account) o;
   // process a
}
else
if (o instanceof Student) {
   Student s = (Student) o;
   // process s
}
```

This can become tedious when retrieving the objects from a container.

13.1.4 A Linked List with Generic Types

The previous section highlighted two problems with using `Object` as the `element` type in the linked list container. These problems are solved in an elegant manner by using Java *generics*. Generics make it possible to parameterize the type of objects to be stored in a container when the container class is being defined. When an instance of the container is created, the programmer can specify the actual type of objects to be stored. The code below shows how the linked list can be parameterized using generics:

```
public class LinkedList<T>
{
   private Node head;
   private int count;

   public LinkedList() {
      head = null;
      count = 0;
   }

   public void addFirst(T element) {
      Node newNode = new Node(element);
      newNode.next = head;
      head = newNode;
      count++;
   }

   public T getFirst() {
      if (count > 0)
         return head.element;
      else
         throw new LinkedListException("List is empty.");
   }

   // other linked list methods

   // Node inner class

   private class Node
   {
      private T element;
      private Node next;

      private Node(T element) {
         this.element = element;
      }
   }
}
```

Notice how the class is parameterized with the type `T` in the first line:

```
public class LinkedList<T>
```

It is normal to use a single letter such as `T` when parameterizing a container class (however, longer names can be used). The type `T` is then used to refer to the objects of the container throughout the class definition. For example, in the `Node` inner class, the `element` of a `Node` is declared to be of type `T`:

```
private T element;
```

In `addFirst()` and other methods, the type `T` is used to specify the input parameters and return values. Thus, no commitment is made as to the exact type of the values that will be stored in the linked list.

When creating an instance of the linked list, the type of object to be stored is specified using angle brackets which are written *before* the argument list of the constructor. For example,

```
LinkedList linkedList = new LinkedList<Account>();
```

Once the linked list is created this way, it can only contain objects of type `Account` since `Account` replaces `T` in the class definition. An attempt to store any other type of object in the linked list will result in a compilation error. Note that by substitutability, the linked list above can store instances of `Account` as well as instances of any of its subclasses. It should also be noted that the call to `getFirst()` now returns an object of the specified type and there is no need to use the `instanceof` operator and casting to obtain the actual object:

```
Account a = linkedList.getFirst();
```

To create a linked list to store `Student` objects, the following code can be used:

```
LinkedList linkedList = new LinkedList<Student>();
```

Thus, by using the parameterized type `T`, the container class only has to be defined once.

The containers in the Java *Collections* framework use generics and thus benefit from the advantages of using parameterized types.

It should be noted that if no type is specified when the container is created, the container behaves as if it stores objects of type `Object` (which is the type of all objects in Java, through substitutability). For example, the container created using the following code will be able to store objects of any type:

```
LinkedList linkedList = new LinkedList();
```

In this case, the compiler generates a warning about type safety.

13.2 Iterators

Different containers store objects in different ways. For example, a linked list consists of a set of nodes where each node stores an object and contains a reference to the next node in the list. A binary tree uses a tree-like structure to store objects ordered in some way. A hash table stores its objects based on a *key* value.

It is highly desirable to enumerate the objects stored in any container without knowing the internal implementation details of that container. This makes it possible to access objects from different containers in a uniform manner. This is the purpose of an iterator. This section describes the Iterator Design Pattern and shows how it is implemented in Java. It also gives an iterator for the linked list described in the previous section.

13.2.1 Iterator Design Pattern

When the *Iterator Design Pattern* is employed, a container object is no longer responsible for accessing and traversing its elements. Instead, access and traversal becomes the responsibility of an iterator object. An iterator object is responsible for keeping track of the current element in the container; thus, an iterator must know which elements have been traversed already.

An iterator defines the interface for accessing the objects from the underlying container. The iterator interface specifies the following methods: `first()`, `next()`, `isDone()`, and `currentItem()`. The `currentItem()` method returns the current element in the container; `first()` initializes the current element to the first element; `next()` advances the current element to the next element; and `isDone()` checks to see if we have advanced beyond the last element, i.e., whether the traversal has come to an end.

13.2.2 Implementation in Java

The Java *Collections* framework defines an `Iterator` interface that accomplishes the design objectives of the Iterator Design Pattern. Thus, classes that implement the `Iterator` interface provide a standard way to traverse and access the objects in a container. The `Iterator` interface is slightly different from the one specified in the Iterator Design Pattern. It is defined as follows using generics:

```
public interface Iterator<T> {
   public abstract boolean hasNext();
   public abstract <T> next();
   public abstract void remove();
}
```

Typically, a container class will contain an inner class that implements the `Iterator` interface. The container class will provide a method `iterator()` that returns an instance of this class to a client object. So, if `c` is an instance of a container, an `Iterator` is obtained as follows:

```
Iterator<Account> i = c.iterator();
```

Notice the use of generics to specify the type of objects which will be iterated. This is the type of object which will be returned by the `next()` method. If no type is specified, the `next()` method will return objects of type `Object`; thus, type checking and casting will be necessary to obtain the actual objects from the container.

Once an instance of an `Iterator` is obtained, its `hasNext()` method can be invoked to determine if there are more elements to be enumerated. This method returns `true` if there are more elements to enumerate, and `false` if all the elements have already been returned. The `next()` method returns the next element in the container. These two methods make it easy to loop through the elements of a container with code such as the following:

```
while (i.hasNext())    // if more elements to enumerate
   process(i.next());  // get next object from iterator and process
```

The `remove()` method removes the object most recently returned by `next()` from the collection. However, support for `remove()` is optional; if an `Iterator` does not implement the `remove()` method, it should throw an `UnsupportedOperationException` when the method is invoked. While iterating through a container, the only way the container can be modified is by calling the `remove()` method of `Iterator`.

It should be noted that many of the concrete collection classes in the Java *Collections* framework provide an `iterator()` method. These include `ArrayList, Vector,` and `TreeSet`.

13.2.3 Linked List Iterator

An iterator for a container is typically implemented as an inner class that implements the `Iterator` interface. The code for an iterator class for `LinkedListGeneric` is given below.

```
public class ListIterator implements Iterator<T>
{
   private Node current;

   private ListIterator (Node head) {
      current = head;
   }

   public boolean hasNext() {
      return (current != null);
   }
   public T next() {
      T save = current.element;
      current = current.next;
      return save;
   }

   public void remove() {
      throw new UnsupportedOperationException
         ("ListIterator does not support remove operation.");
   }

}
```

When an instance of `ListIterator` is created, the current element is positioned at the head of the list. Whenever `next()` is invoked, the current element advances to the next element in the linked list. The `hasNext()` method returns `true` as long as the current element is referring to an actual node of the linked list. When the end of the list is reached, `hasNext()` returns `false`.

The `iterator()` method of the linked list class creates and returns an instance of `ListIterator` when it is called:

```
public Iterator<T> iterator() {
   return new ListIterator(head);
}
```

Client objects can then use the methods of the `Iterator` interface to iterate over the elements in the linked list.

There are two ways to iterate over the objects in a container. The first way is to iterate on the objects exactly as they are stored in the container. `ListIterator` takes this approach. The second way is to copy the object references to another container (e.g., an array) and then iterate over the copied objects in the second container. This approach can be expensive due to copying of objects.

13.2.4 *foreach* Statement

Java provides a *foreach* statement which simplifies the traversal of an `Iterator`. Indeed, code is no longer required to declare an `Iterator` object and to manipulate the `Iterator` using the `hasNext()` and `next()` methods..

Consider the following code which inserts a few `Account` objects in an instance of the `LinkedListGeneric` class:

```
LinkedListGeneric<Account> linkedList =
   new LinkedListGeneric<Account>();

Account a1 = new Account(10, 1000.00);
Account a2 = new Account(20, 2000.00);
Account a3 = new Account(30, 3000.00);

linkedList.addFront(a1);
linkedList.addFront(a2);
linkedList.addFront(a3);
```

A *foreach* statement which traverses the linked list and prints out the information on each `Account` object is given below:

```
for (Account a : linkedList)
   System.out.println(a.toString());
```

The *foreach* statement says that for each `Account` object `a` in `linkedList`, display the contents of `a` using its `toString()` method. In order for this to work, the `LinkedListGeneric` class must implement the `Iterable<T>` interface. The `Iterable<T>` interface only declares one method `iterator()`:

```
Iterator<T> iterator();
```

Since this method was already implemented in `LinkedListGeneric` through the `ListIterator` inner class, all that has to be done is to indicate that it implements `Iterable<T>`:

```
public class LinkedListGeneric<T> implements Iterable<T> {
   :
}
```

Note that the *foreach* statement still uses the `Iterator` object in the background. All it does is simplify the code to declare and traverse the `Iterator`.

13.3 The Java *Collections* Framework

The Java *Collections* framework defines two types of containers: a `Collection`, which is a group of objects and a `Map`, which is a set of mappings between objects. Different types of containers in the *Collections* framework have different properties. For example, a `Set` is a type of `Collection` in which there are no duplicates, and a `List` is a `Collection` in which the elements are ordered in a sequence. This section describes the interfaces and classes in the Java *Collections* framework.

13.3.1 Interfaces in the Java *Collections* Framework

The Java *Collections* framework contains a set of interfaces such as `Collection`, `Set`, `List`, and `Map` that are used as a basis for creating concrete collection classes such as `ArrayList`, `LinkedList`, `HashSet`, and `TreeMap`. A class diagram showing the interfaces in the Java *Collections* framework that will be discussed in this chapter is given in Figure 13.2 below. It is important to understand the features of each interface in the diagram before studying the concrete classes that are derived from them. These interfaces will be described next.

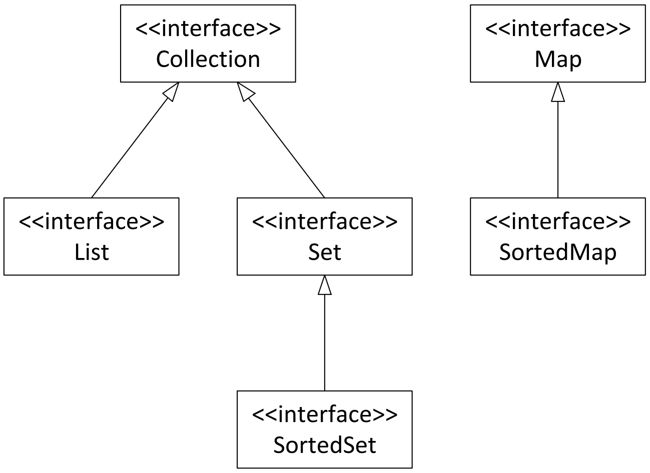

Figure 13.2: Interfaces in the Java *Collections* Framework

13.3.1.1 The `Collection` Interface

The `Collection` interface represents a group or collection of objects. The objects may or may not be ordered, and the collection may or may not contain duplicate objects. The `Collection` interface is not usually implemented directly. Instead, most concrete collection classes implement one of the more specific sub-interfaces. For example, `Set`, an ordered collection that does not allow duplicates, and `List`, an ordered collection that does allow duplicates, are both sub-interfaces of `Collection`. Some of the methods declared in the

`Collection` interface are listed in Table 13.1. Note that all the methods are `public` and `abstract`.

Method	*Description*
boolean add(E o)	Inserts the object of the specified type into the collection; returns `true` if the object was added to the collection; otherwise returns `false`.
boolean addAll(Collection c)	Inserts all the objects from the specified collection into the current collection.
void clear()	Removes all the elements from the collection.
boolean contains (Object o)	Returns `true` if the specified object is present in the collection and `false`, otherwise.
boolean isEmpty()	Returns `true` if there are no elements in the collection and `false`, otherwise.
boolean remove(Object o)	Deletes the specified object from the collection.
int size()	Returns the number of elements currently in the collection.

Table 13.1: Some Methods of the `Collection` Interface

It should be observed that the arguments for methods such as `contains()` and `remove()` are of type `Object`. The `contains()` and `remove()` methods use the `equals()` method of the `Object` class to determine if two objects are the same. If the `equals()` method of `Object` is not overridden in the class of objects being stored in the collection, equality is based on object references and not on the contents of the objects being compared.

13.3.1.2 The `List` Interface

The `List` interface represents an ordered collection of objects (also known as a sequence). Users of the `List` interface have precise control over where in the list each element is inserted. Each element in a `List` has an index, or position, in the list, and elements can be inserted, queried, and removed by index. The first element of a `List` has an index of zero. The last element in a list has an index of `size()-1`. It should be noted that a `List` typically allows duplicate elements, unlike a `Set`.

In addition to the methods declared by its inherited interface, `Collection`, `List` declares a number of methods for working with its indexed elements. Some of these are listed in Table 13.2.

Method	*Description*
void add(int index, E element)	Inserts the object supplied as an argument in position `index` of the `List`. The other elements in the `List` are shifted down one position.
E get(int index)	Returns the element at position `index` of the `List`.
int indexOf(Object o)	Returns the position of the object supplied as an argument in the `List`, or –1 if it does not find a match. `indexOf()` uses the `equals()` method of the contained objects to check for equality with the object supplied as an argument.
E remove(int index)	Removes the object at position `index` of the `List`.
E set(int index, E element)	Inserts the object supplied as `element` in position `index` of the `List`, overwriting the element that was there previously, if any. It returns the element that was previously stored at that position; otherwise, it returns `null`.

Table 13.2: Some Methods of the `List` Interface

13.3.1.3 The `Set` and `SortedSet` Interfaces

The `Set` interface represents an unordered collection of objects that contains no duplicate elements. That is, a `Set` cannot contain two elements, `e1` and `e2`, where `e1.equals(e2)`, and it can contain at most one `null` element. The `Set` interface declares the same methods as its super-interface, `Collection`. It does not allow the `add()` and `addAll()` methods to add duplicate elements to the `Set`. If a `Set` already contains the element being added, its `add()` and `addAll()` methods return `false`.

In order for a `Set` to determine if it already contains an element, it uses the `equals()` method of `Object` to check if this element is equal to an existing element in the `Set`. If the objects in the `Set` do not override the `equals()` method, equality is based on object identifiers. Thus, two objects with exactly the same contents will not be equal and the `Set` will not treat them as duplicates. It is therefore necessary to override the `equals()` method of the objects that will be inserted into the `Set` so that the comparison will be based on the contents of the objects rather than object identifiers.

The `SortedSet` interface is a `Set` that sorts its elements and guarantees that its `iterator()` method returns an `Iterator` that enumerates the elements of the set in sorted order. It declares a few methods of its own such as `first()` and `last()` which return the lowest and highest elements in the set, respectively (as determined by the sort order).

13.3.1.4 The `Map` and `SortedMap` Interface

The `Map` interface represents a collection of mappings between *key* objects and *value* objects. Hash tables are examples of maps. The set of *key* objects in a `Map` must not have any duplicates. However, the collection of *value* objects may contain duplicates. Table 13.3 contains a list of some of the methods in the `Map` interface.

Method	*Description*
boolean containsKey(Object key)	Returns `true` if the `Map` contains a mapping for the specified `key` and `false`, otherwise.
V get(Object key)	Returns the `value` object associated with the specified `key` or `null` if there is no mapping for the `key`.
Set<E> keySet()	Returns a `Set` of all the `key` objects in the `Map`.
V put(K key, V value)	Creates a `key/value` mapping in the `Map`. If the `key` already exists in the `Map`, `put()` replaces the `value` currently in the `Map` with the `value` supplied as an argument and returns the `value` replaced; otherwise, it returns the `value`.
Collection<V> values()	Returns a `Collection` of all the `value` objects in the `Map`.

Table 13.3: Some Methods of the `Map` Interface

To ensure that the set of `key` objects in a `Map` does not contain duplicates, it is important that the `key` objects override the `equals()` method of the `Object` class, based on the contents of the `key`. Note that duplicates depend on how the `equals()` method is defined. If the `equals()` method from `Object` is not overridden, there is a duplicate if two objects references are the same. Two objects might have the same set of values for the attributes but will not be considered to be duplicates if equality is based on object references. To ensure that equality is based on attribute values, the `equals()` method of the `key` object should be overridden.

The `SortedMap` interface represents a `Map` object that keeps its set of `key` objects in sorted order. Its `keySet()` and `values()` methods inherited from `Map` return collections that can be iterated in sorted order of the `key`. It also declares methods of its own such as `firstKey()` and `lastKey()` that return the lowest and highest `key` values in the `SortedMap`.

13.3.2 Classes in the Java Collections Framework

The interfaces in the *Collections* framework have several methods that can be easily implemented. These interfaces are partially implemented in five abstract classes so that concrete collection classes can inherit these methods as a starting point. The five abstract classes are: `AbstractCollection`, `AbstractList`, `AbstractSequentialList`, `AbstractSet`, and `AbstractMap`. A concrete collection class can be implemented by inheriting from one of these abstract classes.

The class diagram in Figure 13.3 shows the concrete classes that are discussed in this chapter and their relationship to the abstract classes. The concrete classes are shaded in the diagram. Note that in addition to inheriting from the abstract classes shown, a concrete class will typically implement one of the interfaces in the Java *Collections* framework.

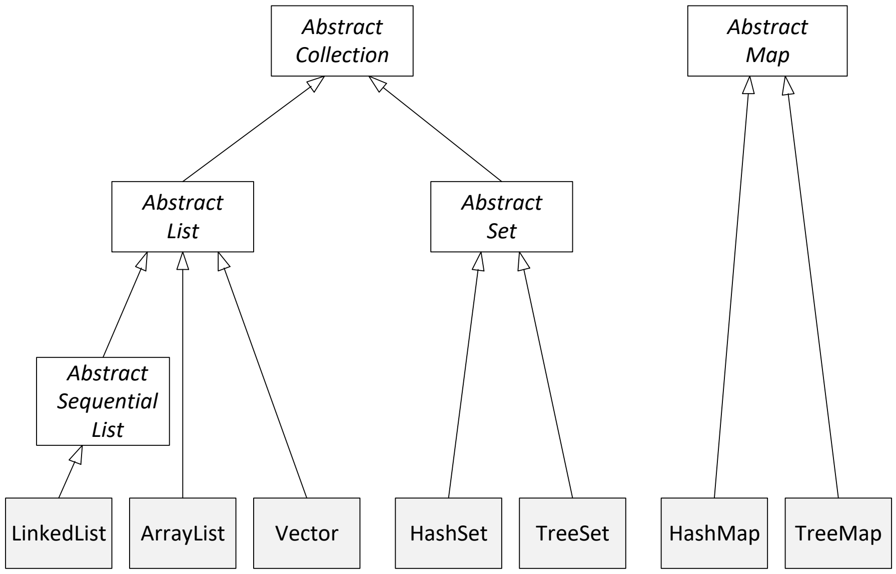

Figure 13.3: Classes in the Java *Collections* Framework

13.3.2.1 LinkedList

A `LinkedList` is a `List` implementation based on an underlying doubly linked list. Since it is implemented using a linked list data structure, each node in the list is connected to its immediate successor and predecessor. This makes its `add()` and `remove()` methods more efficient than an `ArrayList` since an `ArrayList` shifts its elements upwards or downwards whenever a new element is inserted or removed. However, the `get()` and `set()` methods of a `LinkedList` are substantially less efficient than the corresponding `get()` and `set()` methods of `ArrayList` and `Vector` (see next section), since they require traversal of the list from the beginning to find the element at the specified index.

13.3.2.2 ArrayList and Vector

An `ArrayList` is a general-purpose `List` implementation based on an array (that is re-created as necessary as the list grows or shrinks). Since an `ArrayList` is based on an array, its `get()` and `set()` methods are very efficient. A `Vector` is similar to an `ArrayList`, except that its methods are guaranteed that no concurrency control problems will arise. There are additional methods in `Vector` that have the same functionality as some of the methods in `List` described above, to maintain compatibility with earlier versions of Java. These include methods such as `elementAt()` and `setElementAt()`.

An `ArrayList` can be used to store a collection of `Account` objects as follows:

```
ArrayList<Account> accounts = new ArrayList<Account>();
                      // create instance of ArrayList

Account a1 = new Account(10, 1000.00);
Account a2 = new Account(20, 2000.00);
                      // create two Account objects

accounts.add(a1);     // insert a1 into collection
accounts.add(a2);     // insert a2 into collection
```

Since an `ArrayList` implements the `List` interface (which in turn, extends the `Collection` interface), the client of an `ArrayList` can use all the index-based methods of the `List` interface as well as all the methods of the `Collection` interface. A client can choose any method from the `ArrayList` that is appropriate for its needs. For, example, if a client wishes to remove the

`Account` instance, `a`, at position `i` of the `ArrayList`, it could use the index-based `remove()` method of `List`:

```
accounts.remove(i);   // remove Account at position i
```

However, it could also use the `remove()` method from the `Collection` interface:

```
accounts.remove(a);   // remove Account referred to by a
```

The only difference is that the index-based `remove()` will be faster than the `remove()` in `Collection` since there is no need to search for the given object in the `ArrayList`.

Since `ArrayList` implements the `Collection` interface, the following declaration can also be used:

```
Collection<Account> accounts = new ArrayList<Account>();
```

This means that an instance of an `ArrayList` can be used wherever an object of type `Collection` is expected. It is preferable to specify the general type `Collection` in method signatures rather than specific types such as `ArrayList` and `TreeSet` since it allows the signatures to stay the same, even though the underlying implementation of the collection may change for some reason (e.g., replacing an `ArrayList` with a `TreeSet`). However, when this is done, clients will only be able to use the methods of the `Collection` interface.

Before concluding this section, it should be noted that the code given above could be used without change if a `Vector` was used to store the accounts instead of an `ArrayList`, since both `Vector` and `ArrayList` implement the `List` interface.

13.3.2.3 `HashSet`

The `HashSet` class implements the `Set` interface using an internal hash table. Because `HashSet` is based on a hash table, its `add()`, `remove()`, and `contains()` methods are very efficient. However, a `HashSet` makes no guarantee about the order in which the set elements are enumerated by the `Iterator` returned by the `iterator()` method. In the previous section, an example was given showing how to insert elements in an `ArrayList`. No change to this code is required when a `HashSet` is used.

The `Object` class provides a `hashCode()` method that generates a hash code for an object based on its memory address. The `HashSet` uses this method to determine where to place an object in the hash table. However, the hashCode() method of the Object class is generally not very useful since it would generate different hash codes for objects with identical contents. Thus, in order to store objects efficiently in a HashSet, the preferred technique is to override the hashCode() method to generate a (possibly) unique hash code based on the contents of the objects.

The following code is a fragment of an Account class that overrides the hashCode() method of Object:

```
public class Account {
   private int num;
   private double balance;

   // usual Account methods

   public int hashCode() {
      return 13 * num;
   }
}
```

Many of the built-in classes in Java override the hashCode() method to generate more useful hash codes. These hash codes are normally derived from the contents of the objects.

Since a HashSet is a Set, it does not allow duplicates. It is useful to understand how a duplicate is detected. Consider two objects of the Account class which are created as follows:

```
Account a1 = new Account(10, 1000.00);
Account a2 = new Account(10, 1000.00);
```

Notice that a1 and a2 have the same value for number (which is the primary key for Account objects). They also have the same value for the balance attribute. So, a2 can be considered a duplicate of a1.

Now, suppose that a1 and a2 are inserted in the HashSet using its add() method. If the Account class does not override the hashCode() method, there is a strong possibility that a1 and a2 will hash to different locations in the HashSet since the hash code is based on the memory address of a1 and a2. Even if the Account class overrides the equals() method in Object which specifies that two Account objects are equal if they have the same number, a1 and a2 will still hash to different locations and will be both inserted in the HashSet. In other words, it is important to override both the hashCode()

method and the equals() method to ensure that duplicates are not inserted in a HashSet.

13.3.2.4 TreeSet

A TreeSet implements the Set interface using an internal red-black tree so that its elements are maintained in sorted order. Every time an element is added to a TreeSet, it is placed in a position determined by its sort order. Therefore, its iterator() method returns an Iterator that traverses the elements in the TreeSet in sorted order.

Consider the following code:

```
TreeSet<String> names = new TreeSet<String>();

names.add("Shellyann");
names.add("Diana");
names.add("Keshav");

Iterator <String> i = names.iterator();
while (i.hasNext())
   System.out.println(i.next());
```

When run, this code will generate the names in sorted order as follows:

```
Diana
Keshav
Shellyann
```

In order to determine the sort order of the elements in a TreeSet, the objects to be inserted must implement the Comparable interface[1]. In other words, they must supply an implementation of the compareTo() method which specifies how to order two elements in the TreeSet.

It should be noted that since the Collection interface inherits from the Iterable interface, the *foreach* statement can be used to traverse the TreeSet as follows:

```
for (String n : names)
   System.out.println(n);
```

[1] Later in this chapter, an alternative approach to using the *Comparable* interface is discussed.

The *foreach* statement can also be used to traverse the other collections that implement the `Collection` interface.

13.3.2.5 `HashMap`

The `HashMap` class implements the `Map` interface using an internal hash table. Since it is based on a hash table, its `get()` and `put()` methods are very efficient. The following code shows how a `HashMap` can be used for storing accounts:

```
HashMap <Integer, Account> accounts =
   new HashMap<Integer, Account>();

Account a1 = new Account(10, 1000.00);
Account a2 = new Account(20, 2000.00);
Account a3 = new Account(30, 3000.00);

accounts.put(new Integer(a1.getNum()), a1);
accounts.put(new Integer(a2.getNum()), a2);
accounts.put(new Integer(a3.getNum()), a3);
```

In this example, the account number is used as the *key* (an `Integer` object) and the entire account object (`Account`) is used as the value in the `HashMap`. However, since the account number is of the primitive type `int`, it cannot be used directly, since the `put()` method expects the *key* to be of type `Integer`. So, the account number is first converted to an `Integer` object (which overrides the `hashCode()` and `equals()` methods of `Object`, like many of the other Java classes).

Figure 13.4 is a diagram showing how a `HashMap` can be used to store `Account` objects. The *key* is the account `number` (converted to an `Integer` object) and the *value* is the `Account` object itself.

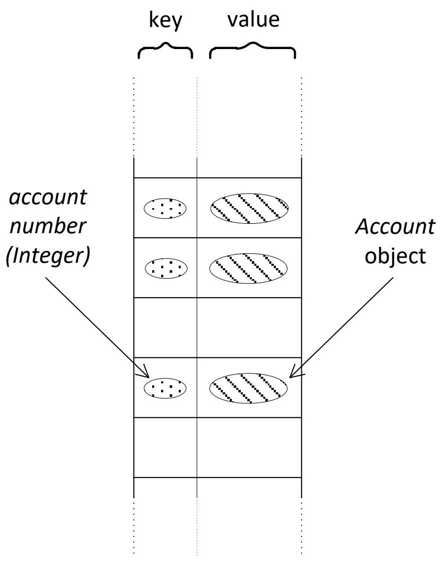

Figure 13.4: HashMap of `Account` Objects

It should be noted that there is no problem with having a duplicate *value* in the `HashMap` (or in a `TreeMap` for that matter), as long as it is associated with a different *key*. For example,

```
accounts.put(new Integer(40), a3);
```

To find the account in the `HashMap` with the account number 10, the following code can be used:

```
Account a = accounts.get(new Integer(10));
```

If there is no element in the `HashMap` with the given *key*, the `get()` method returns `null`.

13.3.2.6 `TreeMap`

The `TreeMap` class implements the `SortedMap` interface using an internal red-black tree data structure. It guarantees that the *keys* and *values* of the mapping can be enumerated based on the order of the *keys*. The objects being used as *keys* in a `TreeMap` must implement the `Comparable` interface.

Consider the following example:

```
TreeMap<Integer, Account> accounts =
   new TreeMap<Integer, Account>();

Account a1 = new Account(10, 1000.00);
Account a2 = new Account(40, 4000.00);
Account a3 = new Account(30, 3000.00);

accounts.put(new Integer(a1.getNum()), a1);
accounts.put(new Integer(a2.getNum()), a2);
accounts.put(new Integer(a3.getNum()), a3);

Set<Integer> keys = accounts.keySet();
Iterator<Integer> i = keys.iterator();

while (i.hasNext()) {
   Integer key = i.next();
   System.out.println(key.toString());
}
```

When run, the code is guaranteed to generate a list of the *keys* in sorted order (based on the implementation of the `compareTo()` method in the `Integer` class):

```
10
30
40
```

This is different from when a `HashMap` is used since a `HashMap` does not guarantee how the *keys* will be ordered.

Next, consider what happens when the following code is executed on a `TreeMap` of `Account` objects:

```
Collection<Account> c = accounts.values();
Iterator<Account> i = c.iterator();

while (i.hasNext()) {
   Account a = i.next();
   System.out.println(a.toString());
}
```

This code will generate a listing of the accounts, in the order `a1`, `a3`, `a2`. This is because the *values* are sorted based on the *key*.

13.4 The `Comparable` Interface (Generics Version)

In the previous chapter, it was shown how the `Comparable` interface can be implemented in the `Account` class. The generics version of the implementation will now be presented. First, the `Account` class must indicate that it implements the `Comparable<T>` interface, replacing `T` with the type `Account`:

```
public class Account implements Comparable<Account>
{
   :
}
```

Next, the `compareTo()` method is implemented. The implementation is very similar to the one previously described except that it is not necessary to use `instanceof` and casting to retrieve the `Account` object passed as an argument. This is because the parameter of the `compareTo()` method is already specified to be of type `Account`. The code is given below.

```
public int compareTo(Account a) {
   if (this.number < a.number)
      return -1;
   else
   if (this.number == a.number)
      return 0;
   else
      return 1;
}
```

13.5 The `Comparator` Interface

A `TreeSet` requires its elements to implement the `Comparable` interface. Similarly, a `TreeMap` requires its keys to implement the `Comparable` interface. One limitation of this approach is that a particular class can only implement the `Comparable` interface once. So based on this approach, it is not possible to store, say, `Account` objects in one `TreeSet` sorted by `number` and in another `TreeSet`, sorted by `balance`. However, there is another approach that can be used to specify multiple sort orders. This approach requires the use of the `Comparator` interface (from `java.util.*`).

Like the `Comparable` interface, the `Comparator` interface has a single method:

```
public interface Comparator<T> {
   public int compare (T o1, T o2);
}
```

The `compare()` method is similar to the `compareTo()` method except that both objects to be compared are supplied as arguments to the method.

To specify a sort order, a class must be created that implements the `Comparator` interface. For example, to specify a sort order for `Account` objects based on account numbers, the following class can be created:

```
public class AccountComparatorNum implements Comparator<Account> {
   public int compare(Account a1, Account a2) {
      return (a1.getNum() - a2.getNum());
   }
}
```

To specify a sort order for `Account` objects based on account balances, the following class can be created:

```
public class AccountComparatorBalance implements Comparator<Account>
{
   public int compare(Account a1, Account a2) {
      return (int) (a1.getBalance() - a2.getBalance());
   }
}
```

Collection classes that depend on a sort order (such as `TreeSet` and `TreeMap`) provide a constructor that accepts a `Comparator` object. To specify that the elements of a `TreeSet` are to be sorted by account number, an instance of `AccountComparatorNum` is created and supplied to the constructor of the `TreeSet` as an argument:

```
Comparator<Account> comparator = new AccountComparatorNum();
TreeSet<Account>accounts = new TreeSet<Account>(comparator);
```

If instead, the accounts are to be sorted by account balance, an instance of `AccountComparatorBalance` is created and supplied to the constructor:

```
Comparator<Account> comparator = new AccountComparatorBalance();
TreeSet<Account> accounts = new TreeSet<Account>(comparator);
```

Thus, a `Comparator` class can be written to define a particular sort order for a class of objects. An instance of the `Comparator` is supplied when creating a collection. This approach is not restricted to collection classes alone. A class that depends on its objects implementing the `Comparable` interface could provide a constructor allowing a `Comparator` to be specified at run-time. In this way, clients can specify one of several sort orders at run-time, by creating and sending an appropriate `Comparator` object.

Both the `AccountComparatorNum` and the `AccountComparatorBalance` classes have no attributes. These classes provide a single method to perform the comparison based on the `Comparator` interface. For that reason, an instance of a class such as `AccountComparatorNum` is known as a *function object*. However, more flexibility can be achieved if instance variables are used. For example, the `AccountComparatorBalance` class can use an instance variable, *ascending*, to specify whether the accounts are to be sorted in ascending order or descending order by balance. A client can then specify a value for this attribute at run-time, according to its needs.

13.6 Which Collection to Use?

This chapter has discussed several concrete collection classes, each with their own features and performance characteristics. In an object-oriented application, you must choose the collection classes that best meet your needs. All the collection classes allow objects to be stored, some based on sorted order, while others are based on no particular order. Some allow index-based access while others enable random access based on hash codes.

Before concluding this discussion on the classes of the Java *Collections* framework, a brief comparison will be given of the performance characteristics of the different containers.

For insertion and deletion of elements, a `LinkedList` is more efficient than an `ArrayList` or a `Vector`. However, for index-based retrieval, the latter two are more efficient and the operation takes place in constant time. The performance of index-based retrieval on a `LinkedList` is of the order n ($O(n)$).

Inserting an element in a `HashSet` or `HashMap` takes place in constant time compared to a `TreeSet` or `TreeMap` where the time complexity is of the order $\log_2 n$ ($O(\log_2 n)$), where n is the number of elements already in the `TreeSet`. Searching for an element in a `HashSet` or `HashMap` also takes place in constant time compared to a `TreeSet` or `TreeMap` where it takes place in ($O(\log_2 n)$ time. So, generally, a `HashSet` or a `HashMap` should be used instead of a `TreeSet` or a `TreeMap`, unless there is a need to maintain elements in sorted order.

Exercises

1. Explain how the generics feature in Java solves the problem of generalizing the type of objects that can be stored in a container while providing type safety at the same time.

2. In a certain human resource application, it is necessary to generate reports of employees sorted in different ways. For example, the employees sometimes need to be sorted by their `last name` and sometimes by their `position`. An `Employee` class represents the concept of an employee in the application. Explain how the different sort orders can be implemented in the application. How can ascending and descending sort orders be accommodated?

3. A class `C` does not override the `equals()` method of the `Object` class but it overrides the `hashCode()` method by generating a value based on the attributes of the class. Two instances of `C` are created where the values of the attributes are the same. Explain what happens if an attempt is made to insert both instances of `C` in a `HashSet`.

4. Explain how you would choose between a `TreeMap` and a `HashMap` for storing a certain collection of objects.

5. Suppose you need to store a collection of objects for which index-based retrieval is important. Compare the use of an `ArrayList`, a `LinkedList`, and a `HashSet` for storing the objects.

6. An `ArrayList` implements the `List` interface which extends the `Collection` interface. How does this affect the operations that a client can perform on an `ArrayList`? Which operations will be more efficient?

7. You are planning to write a new container class, `C`, which will store collections of objects. Will you use the `Collection` interface, the `Map` interface, or neither? Give reasons for your answer.

8. Suppose you have written a collection class, `C`. You would like clients to be able to use the *foreach* statement to traverse the collection. Explain the additional features that should be implemented in `C` to accommodate this functionality. Assuming that `Account` objects are stored in an instance of `C`, write the code to find the sum of all the `Account` balances using a *foreach* statement.

9. How do client objects benefit when a collection class provides an `Iterator` object to its underlying elements?

10. Objects of a certain class `C` must be must be stored in a `TreeSet`. Describe two features that must be implemented in `C` in order for instances of `C` to be stored in a `TreeSet`.