Chapter 2

Collections

- COLLECTION INTERFACES
- CONCRETE COLLECTIONS
- THE COLLECTIONS FRAMEWORK
- ALGORITHMS
- LEGACY COLLECTIONS

Object-oriented programming (OOP) encapsulates data inside classes, but this doesn't make the way in which you organize the data inside the classes any less important than in traditional programming languages. Of course, how you choose to structure the data depends on the problem you are trying to solve. Does your class need a way to easily search through thousands (or even millions) of items quickly? Does it need an ordered sequence of elements *and* the ability to rapidly insert and remove elements in the middle of the sequence? Does it need an array-like structure with random-access ability that can grow at run time? The way you structure your data inside your classes can make a big difference when it comes to implementing methods in a natural style, as well as for performance.

This chapter shows how Java technology can help you accomplish the traditional data structuring needed for serious programming. In college computer science programs, a course called *Data Structures* usually takes a semester to complete, so there are many, many books devoted to this important topic. Exhaustively covering all the data structures that may be useful is not our goal in this chapter; instead, we cover the fundamental ones that the standard Java library supplies. We hope that, after you finish this chapter, you will find it easy to translate any of your data structures to the Java programming language.

Collection Interfaces

Before the release of JDK 1.2, the standard library supplied only a small set of classes for the most useful data structures: `Vector`, `Stack`, `Hashtable`, `BitSet`, and the `Enumeration` interface that

provides an abstract mechanism for visiting elements in an arbitrary container. That was certainly a wise choice—it takes time and skill to come up with a comprehensive collection class library.

With the advent of JDK 1.2, the designers felt that the time had come to roll out a full-fledged set of data structures. They faced a number of conflicting design decisions. They wanted the library to be small and easy to learn. They did not want the complexity of the "Standard Template Library" (or STL) of C++, but they wanted the benefit of "generic algorithms" that STL pioneered. They wanted the legacy classes to fit into the new framework. As all designers of collection libraries do, they had to make some hard choices, and they came up with a number of idiosyncratic design decisions along the way. In this section, we will explore the basic design of the Java collections framework, show you how to put it to work, and explain the reasoning behind some of the more controversial features.

Separating Collection Interfaces and Implementation

As is common for modern data structure libraries, the Java collection library separates *interfaces* and *implementations*. Let us look at that separation with a familiar data structure, the *queue*.

A *queue interface* specifies that you can add elements at the tail end of the queue, remove them at the head, and find out how many elements are in the queue. You use a queue when you need to collect objects and retrieve them in a "first in, first out" fashion (see Figure 2–1).

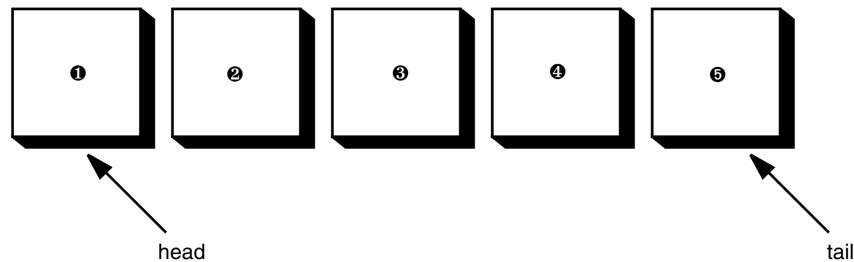


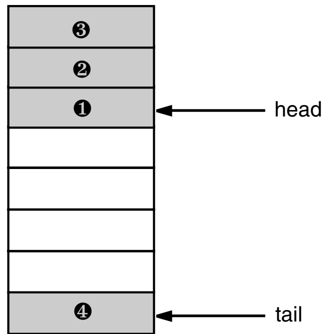
Figure 2–1: A queue

A minimal form of a queue interface might look like this:

```
interface Queue<E> // a simplified form of the interface in the standard library
{
   void add(E element);
   E remove();
   int size();
}
```

The interface tells you nothing about how the queue is implemented. Of the two common implementations of a queue, one uses a "circular array" and one uses a linked list (see Figure 2–2).

NOTE: As of JDK 5.0, the collection classes are generic classes with type parameters. If you use an older version of Java, you need to drop the type parameters and replace the generic types with the `Object` type. For more information on generic classes, please turn to Volume 1, Chapter 13.



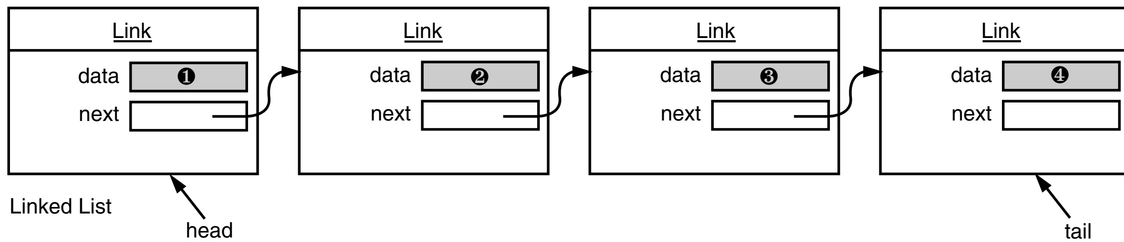


Figure 2–2: Queue implementations

Each implementation can be expressed by a class that implements the `Queue` interface.

```
class CircularArrayQueue<E> implements Queue<E> // not an actual library class
{
   CircularArrayQueue(int capacity) { . . . }
   public void add(E element) { . . . }
   public E remove() { . . . }
   public int size() { . . . }

   private E[] elements;
   private int head;
   private int tail;
}

class LinkedListQueue<E> implements Queue<E>  // not an actual library class
{
   LinkedListQueue() { . . . }
   public void add(E element) { . . . }
   public E remove() { . . . }
   public int size() { . . . }

   private Link head;
   private Link tail;
}
```

NOTE: The Java library doesn't actually have classes named `CircularArrayQueue` and `LinkedListQueue`. We use these classes as examples to explain the conceptual distinction between collection interfaces and implementations. If you need a circular array queue, you can use the `ArrayBlockingQueue` class described in Chapter 1 or the implementation described on page 128. For a linked list queue, simply use the `LinkedList` class—it implements the `Queue` interface.

When you use a queue in your program, you don't need to know which implementation is actually used once the collection has been constructed. Therefore, it makes sense to use the concrete class *only* when you construct the collection object. Use the *interface type* to hold the collection reference.

```
Queue<Customer> expressLane = new CircularArrayQueue<Customer>(100);
expressLane.add(new Customer("Harry"));
```

With this approach if you change your mind, you can easily use a different implementation. You only need to change your program in one place—the constructor. If you decide that a `LinkedListQueue` is a better choice after all, your code becomes

```
Queue<Customer> expressLane = new LinkedListQueue<Customer>();
expressLane.add(new Customer("Harry"));
```

Why would you choose one implementation over another? The interface says nothing about the efficiency of the implementation. A circular array is somewhat more efficient than a linked list, so it is generally preferable. However, as usual, there is a price to pay. The circular array is a *bounded* collection—it has a finite capacity. If you don't have an upper limit on the number of objects that your program will collect, you may be better off with a linked list implementation after all.

When you study the API documentation, you will find another set of classes whose name begins with `Abstract`, such as `AbstractQueue`. These classes are intended for library implementors. To implement your own queue class, you will find it easier to extend `AbstractQueue` than to implement all the methods of the `Queue` interface.

Collection and Iterator Interfaces in the Java Library

The fundamental interface for collection classes in the Java library is the `Collection` interface. The interface has two fundamental methods:

```
public interface Collection<E>
{
   boolean add(E element);
   Iterator<E> iterator();
   . . .
}
```

There are several methods in addition to these two; we discuss them later.

The `add` method adds an element to the collection. The `add` method returns `true` if adding the element actually changes the collection, and `false` if the collection is unchanged. For example, if you try to add an object to a set and the object is already present, then the `add` request has no effect because sets reject duplicates.

The `iterator` method returns an object that implements the `Iterator` interface. You can use the iterator object to visit the elements in the collection one by one.

Iterators

The `Iterator` interface has three methods:

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

By repeatedly calling the `next` method, you can visit the elements from the collection one by one. However, if you reach the end of the collection, the `next` method throws a `NoSuchElementException`. Therefore, you need to call the `hasNext` method before calling `next`. That method returns `true` if the iterator object still has more elements to visit. If you want to inspect all elements in a collection, you request an iterator and then keep calling the `next` method while `hasNext` returns true. For example,

```
Collection<String> c = . . .;
Iterator<String> iter = c.iterator();
while (iter.hasNext())
{
   String element = iter.next();
   do something with element
}
```

As of JDK 5.0, there is an elegant shortcut for this loop. You write the same loop more concisely with the "for each" loop

```
for (String element : c)
{
   do something with element
}
```

The compiler simply translates the "for each" loop into a loop with an iterator.

The "for each" loop works with any object that implements the `Iterable` interface, an interface with a single method:

```
public interface Iterable<E>
{
   Iterator<E> iterator();
}
```

The `Collection` interface extends the `Iterable` interface. Therefore, you can use the "for each" loop with any collection in the standard library.

The order in which the elements are visited depends on the collection type. If you iterate over an `ArrayList`, the iterator starts at index 0 and increments the index in each step. However, if you visit the elements in a `HashSet`, you will encounter them in essentially random order. You can be assured that you will encounter all elements of the collection during the course of the iteration, but you cannot make any assumptions about their ordering. This is usually not a problem because the ordering does not matter for computations such as computing totals or counting matches.

NOTE: Old-timers will notice that the `next` and `hasNext` methods of the `Iterator` interface serve the same purpose as the `nextElement` and `hasMoreElements` methods of an `Enumeration`. The designers of the Java collection library could have chosen to make use of the `Enumeration` interface. But they disliked the cumbersome method names and instead introduced a new interface with shorter method names.

There is an important conceptual difference between iterators in the Java collection library and iterators in other libraries. In traditional collection libraries such as the Standard

Template Library of C++, iterators are modeled after array indexes. Given such an iterator, you can look up the element that is stored at that position, much like you can look up an array element `a[i]` if you have an array index `i`. Independently of the lookup, you can advance the iterator to the next position. This is the same operation as advancing an array index by calling `i++`, without performing a lookup. However, the Java iterators do not work like that. The lookup and position change are tightly coupled. The only way to look up an element is to call `next`, and that lookup advances the position.

Instead, you should think of Java iterators as being *between elements*. When you call `next`, the iterator *jumps over* the next element, and it returns a reference to the element that it just passed (see Figure 2–3).

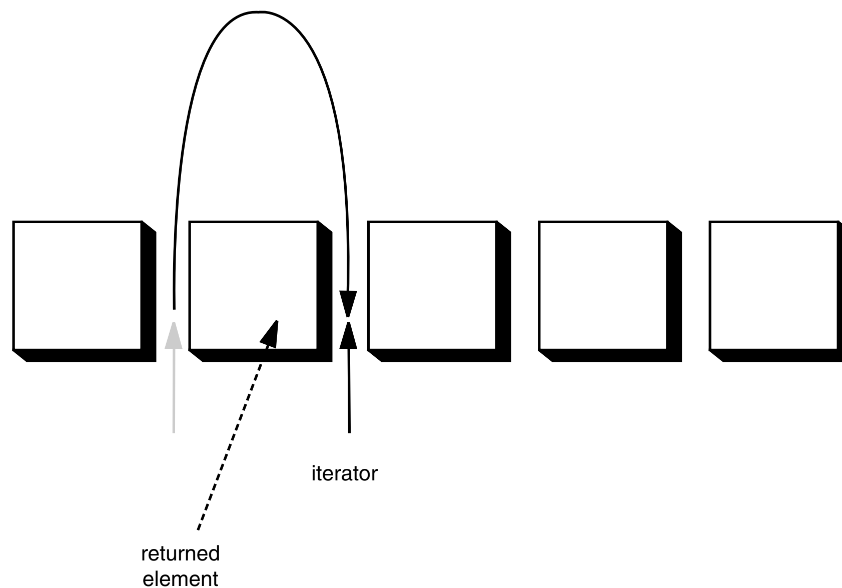


Figure 2–3: Advancing an iterator

NOTE: Here is another useful analogy. You can think of `Iterator.next` as the equivalent of `InputStream.read`. Reading a byte from a stream automatically "consumes" the byte. The next call to `read` consumes and returns the next byte from the input. Similarly, repeated calls to `next` let you read all elements in a collection.

Removing Elements

The `remove` method of the `Iterator` interface removes the element that was returned by the last call to `next`. In many situations, that makes sense—you need to see the element before you can decide that it is the one that should be removed. But if you want to remove an element in a particular position, you still need to skip past the element. For example, here is how you remove the first element in a collection of strings.

```
Iterator<String> it = c.iterator();
it.next(); // skip over the first element
it.remove(); // now remove it
```

More important, there is a dependency between calls to the `next` and `remove` methods. It is illegal to call `remove` if it wasn't preceded by a call to `next`. If you try, an `IllegalStateException` is thrown.

If you want to remove two adjacent elements, you cannot simply call

```
it.remove();
it.remove(); // Error!
```

Instead, you must first call `next` to jump over the element to be removed.

```
it.remove();
it.next();
it.remove(); // Ok
```

Generic Utility Methods

Because the `Collection` and `Iterator` interfaces are generic, you can write utility methods that operate on any kind of collection. For example, here is a generic method that tests whether an arbitrary collection contains a given element:

```
public static <E> boolean contains(Collection<E> c, Object obj)
{
   for (E element : c)
      if (element.equals(obj))
         return true;
   return false;
}
```

The designers of the Java library decided that some of these utility methods are so useful that the library should make them available. That way, library users don't have to keep reinventing the wheel. The `contains` method is one such method.

In fact, the `Collection` interface declares quite a few useful methods that all implementing classes must supply. Among them are:

```
int size()
boolean isEmpty()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
boolean equals(Object other)
boolean addAll(Collection<? extends E> from)
boolean remove(Object obj)
boolean removeAll(Collection<?> c)
void clear()
boolean retainAll(Collection<?> c)
Object[] toArray()
<T> T[] toArray(T[] arrayToFill)
```

Many of these methods are self-explanatory; you will find full documentation in the API notes at the end of this section.

Of course, it is a bother if every class that implements the `Collection` interface has to supply so many routine methods. To make life easier for implementors, the library supplies a class `AbstractCollection` that leaves the fundamental methods `size` and `iterator` abstract but implements the routine methods in terms of them. For example:

```
public abstract class AbstractCollection<E>
   implements Collection<E>
{
   . . .
   public abstract Iterator<E> iterator();

   public boolean contains(Object obj)
   {
      for (E element : c) // calls iterator()
         if (element.equals(obj))
```

```
            return = true;
        return false;
    }
    . . .
}
```

A concrete collection class can now extend the `AbstractCollection` class. It is now up to the concrete collection class to supply an `iterator` method, but the `contains` method has been taken care of by the `AbstractCollection` superclass. However, if the subclass has a more efficient way of implementing `contains`, it is free to do so.

This is a good design for a class framework. The users of the collection classes have a richer set of methods available in the generic interface, but the implementors of the actual data structures do not have the burden of implementing all the routine methods.

java.util.Collection<E> 1.2

- `Iterator<E> iterator()`
  returns an iterator that can be used to visit the elements in the collection.
- `int size()`
  returns the number of elements currently stored in the collection.
- `boolean isEmpty()`
  returns `true` if this collection contains no elements.
- `boolean contains(Object obj)`
  returns `true` if this collection contains an object equal to `obj`.
- `boolean containsAll(Collection<?> other)`
  returns `true` if this collection contains all elements in the other collection.
- `boolean add(Object element)`
  adds an element to the collection. Returns `true` if the collection changed as a result of this call.
- `boolean addAll(Collection<? extends E> other)`
  adds all elements from the other collection to this collection. Returns `true` if the collection changed as a result of this call.
- `boolean remove(Object obj)`
  removes an object equal to `obj` from this collection. Returns `true` if a matching object was removed.
- `boolean removeAll(Collection<?> other)`
  removes from this collection all elements from the other collection. Returns `true` if the collection changed as a result of this call.
- `void clear()`
  removes all elements from this collection.
- `boolean retainAll(Collection<?> other)`
  removes all elements from this collection that do not equal one of the elements in the other collection. Returns `true` if the collection changed as a result of this call.
- `Object[] toArray()`
  returns an array of the objects in the collection.

java.util.Iterator<E> 1.2

- `boolean hasNext()`
  returns `true` if there is another element to visit.

CJ7V2.book Page 93 Wednesday, October 27, 2004 2:32 PM

- `E next()`
  returns the next object to visit. Throws a `NoSuchElementException` if the end of the collection has been reached.
- `void remove()`
  removes the last visited object. This method must immediately follow an element visit. If the collection has been modified since the last element visit, then the method throws an `IllegalStateException`.

Concrete Collections

Rather than getting into more details about all the interfaces, we thought it would be helpful to first discuss the concrete data structures that the Java library supplies. Once we have thoroughly described the classes you might want to use, we will return to abstract considerations and see how the collections framework organizes these classes. Table 2–1 shows the collections in the Java library and briefly describes the purpose of each collection class. (For simplicity, we omit the thread-safe collections that were discussed in Chapter 1.) All classes in Table 2–1 implement the `Collection` interface, with the exception of the classes with names ending in `Map`. Those classes implement the `Map` interface instead. We will discuss the `Map` interface on page 110.

Table 2–1: Concrete Collections in the Java Library

Collection Type	Description	See Page
`ArrayList`	An indexed sequence that grows and shrinks dynamically	101
`LinkedList`	An ordered sequence that allows efficient insertions and removal at any location	93
`HashSet`	An unordered collection that rejects duplicates	101
`TreeSet`	A sorted set	104
`EnumSet`	A set of enumerated type values	116
`LinkedHashSet`	A set that remembers the order in which elements were inserted	115
`PriorityQueue`	A collection that allows efficient removal of the smallest element	109
`HashMap`	A data structure that stores key/value associations	110
`TreeMap`	A map in which the keys are sorted	110
`EnumMap`	A map in which the keys belong to an enumerated type	116
`LinkedHashMap`	A map that remembers the order in which entries were added	115
`WeakHashMap`	A map with values that can be reclaimed by the garbage collector if they are not used elsewhere	114
`IdentityHashMap`	A map with keys that are compared by `==`, not `equals`	117

Linked Lists

We used arrays and their dynamic cousin, the `ArrayList` class, for many examples in Volume 1. However, arrays and array lists suffer from a major drawback. Removing an element from the middle of an array is expensive since all array elements beyond the removed one must be moved toward the beginning of the array (see Figure 2–4). The same is true for inserting elements in the middle.

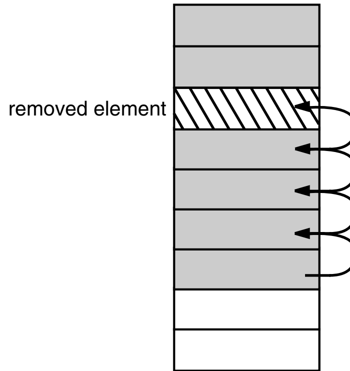


Figure 2–4: Removing an element from an array

Another well-known data structure, the *linked list*, solves this problem. Whereas an array stores object references in consecutive memory locations, a linked list stores each object in a separate *link*. Each link also stores a reference to the next link in the sequence. In the Java programming language, all linked lists are actually *doubly linked*; that is, each link also stores a reference to its predecessor (see Figure 2–5).

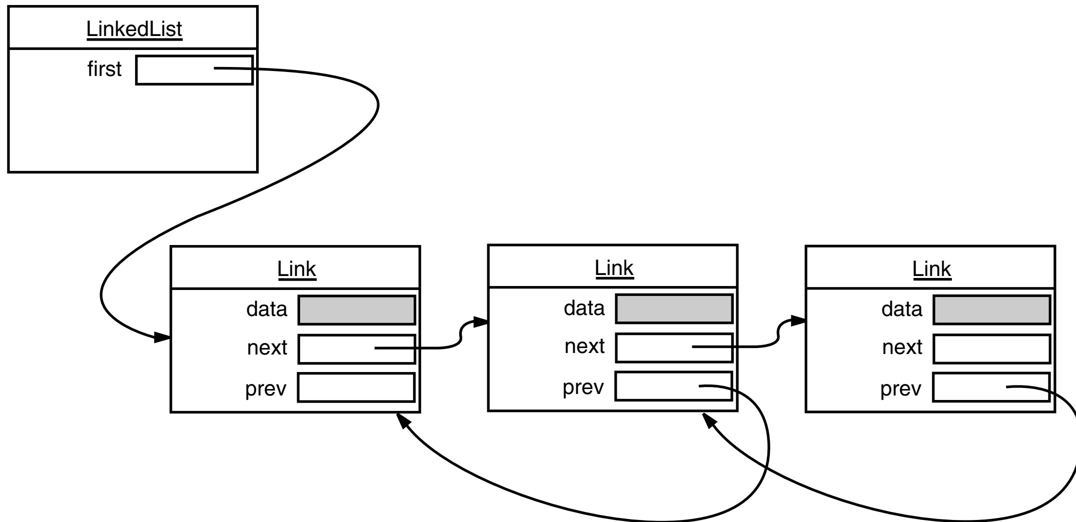


Figure 2–5: A doubly linked list

Removing an element from the middle of a linked list is an inexpensive operation—only the links around the element to be removed need to be updated (see Figure 2–6).

Perhaps you once took a data structures course in which you learned how to implement linked lists. You may have bad memories of tangling up the links when removing or adding elements in the linked list. If so, you will be pleased to learn that the Java collections library supplies a class `LinkedList` ready for you to use.

The following code example adds three elements and and then removes the second one.

```
List<String> staff = new LinkedList<String>(); // LinkedList implements List
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
Iterator iter = staff.iterator();
String first = iter.next(); // visit first element
String second = iter.next(); // visit second element
iter.remove(); // remove last visited element
```

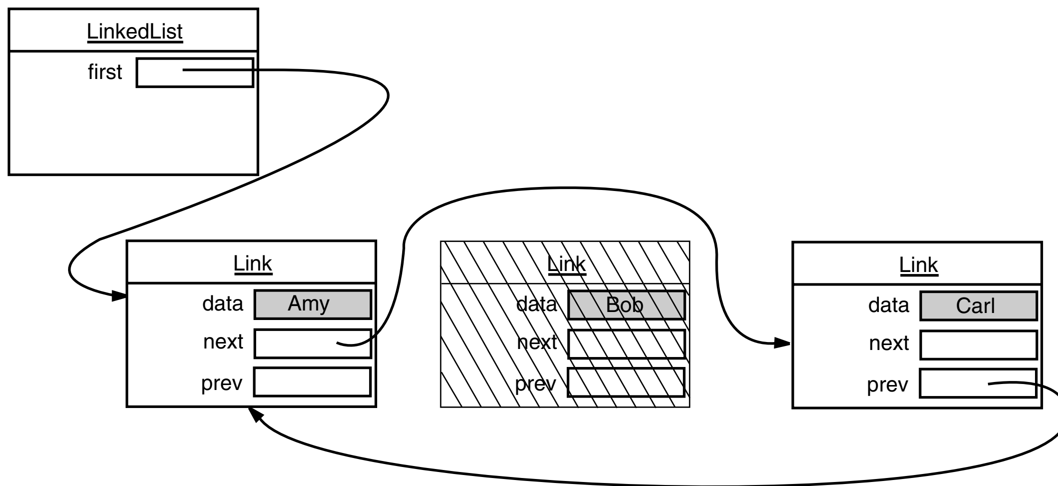


Figure 2–6: Removing an element from a linked list

There is, however, an important difference between linked lists and generic collections. A linked list is an *ordered collection* in which the position of the objects matters. The `LinkedList.add` method adds the object to the end of the list. But you often want to add objects somewhere in the middle of a list. This position-dependent `add` method is the responsibility of an iterator, since iterators describe positions in collections. Using iterators to add elements makes sense only for collections that have a natural ordering. For example, the *set* data type that we discuss in the next section does not impose any ordering on its elements. Therefore, there is no `add` method in the `Iterator` interface. Instead, the collections library supplies a subinterface `ListIterator` that contains an `add` method:

```
interface ListIterator<E> extends Iterator<E>
{
   void add(E element);
   . . .
}
```

Unlike `Collection.add`, this method does not return a `boolean`—it is assumed that the `add` operation always modifies the list.

In addition, the `ListIterator` interface has two methods that you can use for traversing a list backwards.

```
E previous()
boolean hasPrevious()
```

Like the `next` method, the `previous` method returns the object that it skipped over.

The `listIterator` method of the `LinkedList` class returns an iterator object that implements the `ListIterator` interface.

```
ListIterator<String> iter = staff.listIterator();
```

The `add` method adds the new element *before* the iterator position. For example, the following code skips past the first element in the linked list and adds `"Juliet"` before the second element (see Figure 2–7).

```
List<String> staff = new LinkedList<String>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter = staff.listIterator();
iter.next(); // skip past first element
iter.add("Juliet");
```

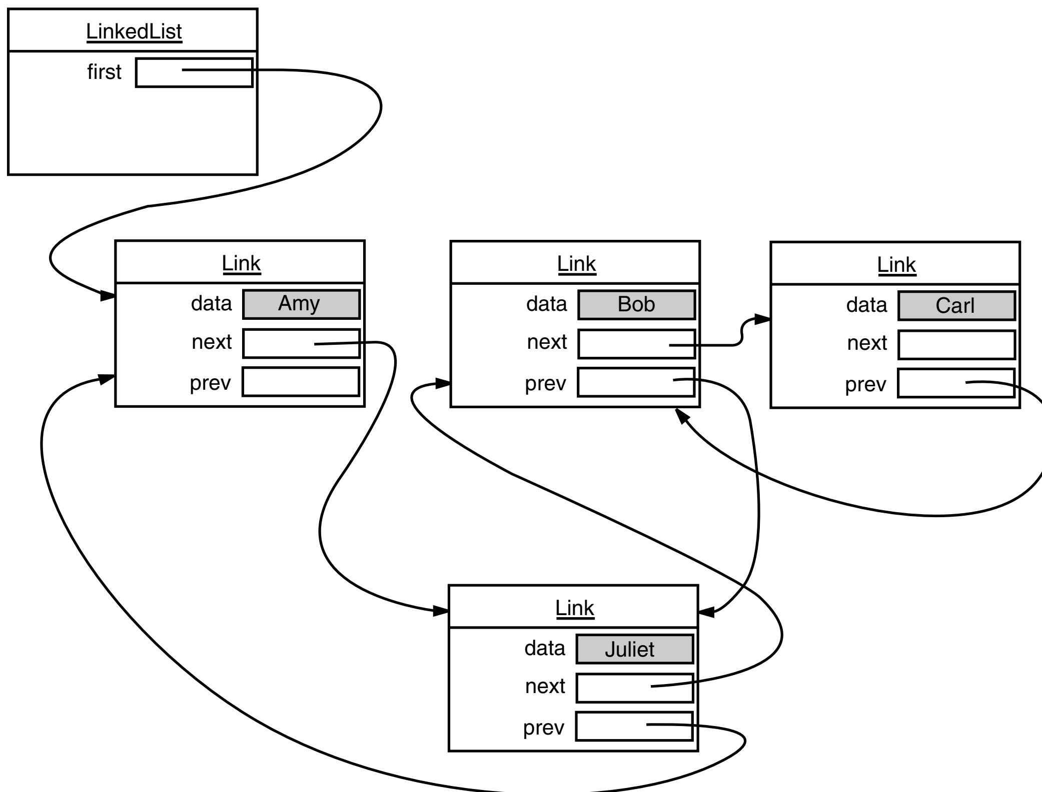


Figure 2–7: Adding an element to a linked list

If you call the `add` method multiple times, the elements are simply added in the order in which you supplied them. They are all added in turn before the current iterator position.

When you use the `add` operation with an iterator that was freshly returned from the `listIterator` method and that points to the beginning of the linked list, the newly added element becomes the new head of the list. When the iterator has passed the last element of the list (that is, when `hasNext` returns `false`), the added element becomes the new tail of the list. If the

linked list has *n* elements, there are *n*+1 spots for adding a new element. These spots correspond to the *n*+1 possible positions of the iterator. For example, if a linked list contains three elements, A, B, and C, then there are four possible positions (marked as |) for inserting a new element:

```
|ABC
A|BC
AB|C
ABC|
```

NOTE: You have to be careful with the "cursor" analogy. The `remove` operation does not quite work like the BACKSPACE key. Immediately after a call to `next`, the `remove` method indeed removes the element to the left of the iterator, just like the BACKSPACE key would. However, if you just called `previous`, the element to the right is removed. And you can't call `remove` twice in a row.

Unlike the `add` method, which depends only on the iterator position, the `remove` method depends on the iterator state.

Finally, a `set` method replaces the last element returned by a call to `next` or `previous` with a new element. For example, the following code replaces the first element of a list with a new value:

```
ListIterator<String> iter = list.listIterator();
String oldValue = iter.next(); // returns first element
iter.set(newValue); // sets first element to newValue
```

As you might imagine, if an iterator traverses a collection while another iterator is modifying it, confusing situations can occur. For example, suppose an iterator points before an element that another iterator has just removed. The iterator is now invalid and should no longer be used. The linked list iterators have been designed to detect such modifications. If an iterator finds that its collection has been modified by another iterator or by a method of the collection itself, then it throws a `ConcurrentModificationException`. For example, consider the following code:

```
List<String> list = . . .;
ListIterator<String> iter1 = list.listIterator();
ListIterator<String> iter2 = list.listIterator();
iter1.next();
iter1.remove();
iter2.next(); // throws ConcurrentModificationException
```

The call to `iter2.next` throws a `ConcurrentModificationException` since `iter2` detects that the list was modified externally.

To avoid concurrent modification exceptions, follow this simple rule: You can attach as many iterators to a collection as you like, provided that all of them are only readers. Alternatively, you can attach a single iterator that can both read and write.

Concurrent modification detection is achieved in a simple way. The collection keeps track of the number of mutating operations (such as adding and removing elements). Each iterator keeps a separate count of the number of mutating operations that *it* was responsible for. At the beginning of each iterator method, the iterator simply checks whether its own mutation count equals that of the collection. If not, it throws a `ConcurrentModificationException`. This is an excellent check and a great improvement over the fundamentally unsafe iterators in the C++ STL framework.

NOTE: There is, however, a curious exception to the detection of concurrent modifications. The linked list only keeps track of *structural* modifications to the list, such as adding and removing links. The `set` method does *not* count as a structural modification. You can attach multiple iterators to a linked list, all of which call `set` to change the contents of existing links. This capability is required for a number of algorithms in the `Collections` class that we discuss later in this chapter.

Now you have seen the fundamental methods of the `LinkedList` class. You use a `ListIterator` to traverse the elements of the linked list in either direction and to add and remove elements.

As you saw in the preceding section, many other useful methods for operating on linked lists are declared in the `Collection` interface. These are, for the most part, implemented in the `AbstractCollection` superclass of the `LinkedList` class. For example, the `toString` method invokes `toString` on all elements and produces one long string of the format `[A, B, C]`. This is handy for debugging. Use the `contains` method to check whether an element is present in a linked list. For example, the call `staff.contains("Harry")` returns `true` if the linked list already contains a string that is equal to the string `"Harry"`.

The library also supplies a number of methods that are, from a theoretical perspective, somewhat dubious. Linked lists do not support fast random access. If you want to see the nth element of a linked list, you have to start at the beginning and skip past the first $n - 1$ elements first. There is no shortcut. For that reason, programmers don't usually use linked lists in programming situations in which elements need to be accessed by an integer index.

Nevertheless, the `LinkedList` class supplies a `get` method that lets you access a particular element:

```
LinkedList<String> list = . . .;
String obj = list.get(n);
```

Of course, this method is not very efficient. If you find yourself using it, you are probably using the wrong data structure for your problem.

You should *never* use this illusory random access method to step through a linked list. The code

```
for (int i = 0; i < list.size(); i++)
   do something with list.get(i);
```

is staggeringly inefficient. Each time you look up another element, the search starts again from the beginning of the list. The `LinkedList` object makes no effort to cache the position information.

NOTE: The `get` method has one slight optimization: If the index is at least `size() / 2`, then the search for the element starts at the end of the list.

The list iterator interface also has a method to tell you the index of the current position. In fact, because Java iterators conceptually point between elements, it has two of them: The `nextIndex` method returns the integer index of the element that would be returned by the next call to `next`; the `previousIndex` method returns the index of the element that would be returned by the next call to `previous`. Of course, that is simply one less than `nextIndex`. These methods are efficient—the iterators keep a count of the current position. Finally, if you have an integer index `n`, then `list.listIterator(n)` returns an iterator that points just before the element with index `n`. That is, calling `next` yields the same element as `list.get(n)`; obtaining that iterator is inefficient.

If you have a linked list with only a handful of elements, then you don't have to be overly paranoid about the cost of the `get` and `set` methods. But then why use a linked list in the first place? The only reason to use a linked list is to minimize the cost of insertion and removal in the middle of the list. If you have only a few elements, you can just use an `ArrayList`.

We recommend that you simply stay away from all methods that use an integer index to denote a position in a linked list. If you want random access into a collection, use an array or `ArrayList`, not a linked list.

The program in Example 2–1 puts linked lists to work. It simply creates two lists, merges them, then removes every second element from the second list, and finally tests the `removeAll` method. We recommend that you trace the program flow and pay special attention to the iterators. You may find it helpful to draw diagrams of the iterator positions, like this:

```
|ACE   |BDFG
A|CE   |BDFG
AB|CE  B|DFG
. . .
```

Note that the call

```
System.out.println(a);
```

prints all elements in the linked list `a` by invoking the `toString` method in `AbstractCollection`.

Example 2–1: LinkedListTest.java

```
1. import java.util.*;
2.
3. /**
4.    This program demonstrates operations on linked lists.
5. */
6. public class LinkedListTest
7. {
8.    public static void main(String[] args)
9.    {
10.      List<String> a = new LinkedList<String>();
11.      a.add("Amy");
12.      a.add("Carl");
13.      a.add("Erica");
14.
15.      List<String> b = new LinkedList<String>();
16.      b.add("Bob");
17.      b.add("Doug");
18.      b.add("Frances");
19.      b.add("Gloria");
20.
21.      // merge the words from b into a
22.
23.      ListIterator<String> aIter = a.listIterator();
24.      Iterator<String> bIter = b.iterator();
25.
26.      while (bIter.hasNext())
27.      {
28.         if (aIter.hasNext()) aIter.next();
29.         aIter.add(bIter.next());
30.      }
31.
32.      System.out.println(a);
33.
```

```
34.       // remove every second word from b
35.
36.       bIter = b.iterator();
37.       while (bIter.hasNext())
38.       {
39.          bIter.next(); // skip one element
40.          if (bIter.hasNext())
41.          {
42.             bIter.next(); // skip next element
43.             bIter.remove(); // remove that element
44.          }
45.       }
46.
47.       System.out.println(b);
48.
49.       // bulk operation: remove all words in b from a
50.
51.       a.removeAll(b);
52.
53.       System.out.println(a);
54.    }
55. }
```

java.util.List<E> 1.2

- `ListIterator<E> listIterator()`
  returns a list iterator for visiting the elements of the list.
- `ListIterator<E> listIterator(int index)`
  returns a list iterator for visiting the elements of the list whose first call to `next` will return the element with the given index.
- `void add(int i, E element)`
  adds an element at the specified position.
- `void addAll(int i, Collection<? extends E> elements)`
  adds all elements from a collection to the specified position.
- `E remove(int i)`
  removes and returns an element at the specified position.
- `E set(int i, E element)`
  replaces the element at the specified position with a new element and returns the old element.
- `int indexOf(Object element)`
  returns the position of the first occurrence of an element equal to the specified element, or –1 if no matching element is found.
- `int lastIndexOf(Object element)`
  returns the position of the last occurrence of an element equal to the specified element, or –1 if no matching element is found.

java.util.ListIterator<E> 1.2

- `void add(E newElement)`
  adds an element before the current position.
- `void set(E newElement)`
  replaces the last element visited by `next` or `previous` with a new element. Throws an `IllegalStateException` if the list structure was modified since the last call to `next` or `previous`.

- `boolean hasPrevious()`
  returns `true` if there is another element to visit when iterating backwards through the list.
- `E previous()`
  returns the previous object. Throws a `NoSuchElementException` if the beginning of the list has been reached.
- `int nextIndex()`
  returns the index of the element that would be returned by the next call to `next`.
- `int previousIndex()`
  returns the index of the element that would be returned by the next call to `previous`.

java.util.LinkedList<E> 1.2

- `LinkedList()`
  constructs an empty linked list.
- `LinkedList(Collection<? extends E> elements)`
  constructs a linked list and adds all elements from a collection.
- `void addFirst(E element)`
- `void addLast(E element)`
  add an element to the beginning or the end of the list.
- `E getFirst()`
- `E getLast()`
  return the element at the beginning or the end of the list.
- `E removeFirst()`
- `E removeLast()`
  remove and return the element at the beginning or the end of the list.

Array Lists

In the preceding section, you saw the `List` interface and the `LinkedList` class that implements it. The `List` interface describes an ordered collection in which the position of elements matters. There are two protocols for visiting the elements: through an iterator and by random access with methods `get` and `set`. The latter is not appropriate for linked lists, but of course `get` and `set` make a lot of sense for arrays. The collections library supplies the familiar `ArrayList` class that also implements the `List` interface. An `ArrayList` encapsulates a dynamically reallocated array of objects.

NOTE: If you are a veteran Java programmer, you may have used the `Vector` class whenever you needed a dynamic array. Why use an `ArrayList` instead of a `Vector`? For one simple reason: All methods of the `Vector` class are *synchronized*. It is safe to access a `Vector` object from two threads. But if you access a vector from only a single thread—by far the more common case—your code wastes quite a bit of time with synchronization. In contrast, the `ArrayList` methods are not synchronized. We recommend that you use an `ArrayList` instead of a `Vector` whenever you don't need synchronization.

Hash Sets

Linked lists and arrays let you specify the order in which you want to arrange the elements. However, if you are looking for a particular element and you don't remember its position, then you need to visit all elements until you find a match. That can be time consuming if the collection contains many elements. If you don't care about the ordering of the elements, then there are data structures that let you find elements much faster. The drawback is that those data structures give you no control over the order in which the elements

appear. The data structures organize the elements in an order that is convenient for their own purposes.

A well-known data structure for finding objects quickly is the *hash table*. A hash table computes an integer, called the *hash code*, for each object. A hash code is an integer that is somehow derived from the instance fields of an object, preferably such that objects with different data yield different codes. Table 2–2 lists a few examples of hash codes that result from the `hashCode` method of the `String` class.

Table 2–2: Hash Codes Resulting from the `hashCode` Function

String	Hash Code
`"Lee"`	76268
`"lee"`	107020
`"eel"`	100300

If you define your own classes, you are responsible for implementing your own `hashCode` method—see Volume 1, Chapter 5 for more information. Your implementation needs to be compatible with the `equals` method: If `a.equals(b)`, then `a` and `b` must have the same hash code.

What's important for now is that hash codes can be computed quickly and that the computation depends only on the state of the object that needs to be hashed, and not on the other objects in the hash table.

A hash table is an array of linked lists. Each list is called a *bucket* (see Figure 2–8). To find the place of an object in the table, compute its hash code and reduce it modulo the total number of buckets. The resulting number is the index of the bucket that holds the element. For example, if an object has hash code 76268 and there are 128 buckets, then the object is placed in bucket 108 (because the remainder 76268 `%` 128 is 108). Perhaps you are lucky and there is no other element in that bucket. Then, you simply insert the element into that bucket. Of course, it is inevitable that you sometimes hit a bucket that is already filled. This is called a *hash collision*. Then, you compare the new object with all objects in that bucket to see if it is already present. Provided that the hash codes are reasonably randomly distributed and the number of buckets is large enough, only a few comparisons should be necessary.

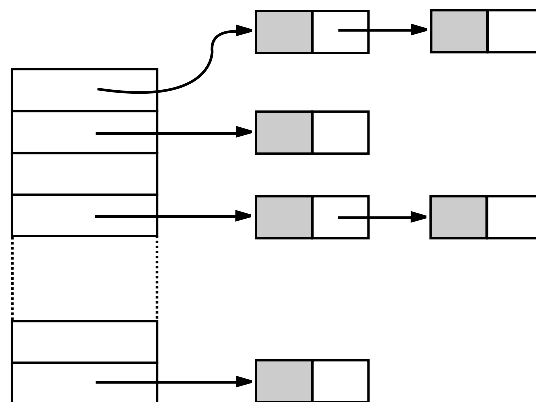


Figure 2–8: A hash table

If you want more control over the performance of the hash table, you can specify the initial bucket count. The bucket count gives the number of buckets that are used to collect objects with identical hash values. If too many elements are inserted into a hash table, the number of collisions increases and retrieval performance suffers.

If you know approximately how many elements will eventually be in the table, then you can set the bucket count. Typically, you set it to somewhere between 75% and 150% of the expected element count. Some researchers believe that it is a good idea to make the bucket count a prime number to prevent a clustering of keys. The evidence for this isn't conclusive, however. The standard library uses bucket counts that are a power of 2, with a default of 16. (Any value you supply for the table size is automatically rounded to the next power of 2.)

Of course, you do not always know how many elements you need to store, or your initial guess may be too low. If the hash table gets too full, it needs to be *rehashed*. To rehash the table, a table with more buckets is created, all elements are inserted into the new table, and the original table is discarded. The *load factor* determines when a hash table is rehashed. For example, if the load factor is 0.75 (which is the default) and the table is more than 75% full, then it is automatically rehashed, with twice as many buckets. For most applications, it is reasonable to leave the load factor at 0.75.

Hash tables can be used to implement several important data structures. The simplest among them is the *set* type. A set is a collection of elements without duplicates. The `add` method of a set first tries to find the object to be added, and adds it only if it is not yet present.

The Java collections library supplies a `HashSet` class that implements a set based on a hash table. You add elements with the `add` method. The `contains` method is redefined to make a fast lookup to find if an element is already present in the set. It checks only the elements in one bucket and not all elements in the collection.

The hash set iterator visits all buckets in turn. Because the hashing scatters the elements around in the table, they are visited in seemingly random order. You would only use a `HashSet` if you don't care about the ordering of the elements in the collection.

The sample program at the end of this section (Example 2–2) reads words from `System.in`, adds them to a set, and finally prints out all words in the set. For example, you can feed the program the text from *Alice in Wonderland* (which you can obtain from `http://www.gutenberg.net`) by launching it from a command shell as

```
java SetTest < alice30.txt
```

The program reads all words from the input and adds them to the hash set. It then iterates through the unique words in the set and finally prints out a count. (*Alice in Wonderland* has 5,909 unique words, including the copyright notice at the beginning.) The words appear in random order.

CAUTION: Be careful when you mutate set elements. If the hash code of an element were to change, then the element would no longer be in the correct position in the data structure.

Example 2–2: SetTest.java

```
1. import java.util.*;
2.
3. /**
4.    This program uses a set to print all unique words in
5.    System.in.
6. */
7. public class SetTest
```

```
8. {
9.    public static void main(String[] args)
10.   {
11.      Set<String> words = new HashSet<String>(); // HashSet implements Set
12.      long totalTime = 0;
13.
14.      Scanner in = new Scanner(System.in);
15.      while (in.hasNext())
16.      {
17.         String word = in.next();
18.         long callTime = System.currentTimeMillis();
19.         words.add(word);
20.         callTime = System.currentTimeMillis() - callTime;
21.         totalTime += callTime;
22.      }
23.
24.      Iterator<String> iter = words.iterator();
25.      for (int i = 1; i <= 20; i++)
26.         System.out.println(iter.next());
27.      System.out.println(". . .");
28.      System.out.println(words.size() + " distinct words. " + totalTime + " milliseconds.");
29.   }
30. }
```

java.util.HashSet<E> 1.2

- `HashSet()`
  constructs an empty hash set.
- `HashSet(Collection<? extends E> elements)`
  constructs a hash set and adds all elements from a collection.
- `HashSet(int initialCapacity)`
  constructs an empty hash set with the specified capacity.
- `HashSet(int initialCapacity, float loadFactor)`
  constructs an empty hash set with the specified capacity and load factor (a number between 0.0 and 1.0 that determines at what percentage of fullness the hash table will be rehashed into a larger one).

java.lang.Object 1.0

- `int hashCode()`
  returns a hash code for this object. A hash code can be any integer, positive or negative. The definitions of `equals` and `hashCode` must be compatible: If `x.equals(y)` is `true`, then `x.hashCode()` must be the same value as `y.hashCode()`.

Tree Sets

The `TreeSet` class is similar to the hash set, with one added improvement. A tree set is a *sorted collection.* You insert elements into the collection in any order. When you iterate through the collection, the values are automatically presented in sorted order. For example, suppose you insert three strings and then visit all elements that you added.

```
SortedSet<String> sorter = new TreeSet<String>(); // TreeSet implements SortedSet
sorter.add("Bob");
sorter.add("Amy");
sorter.add("Carl");
for (String s : sorter) System.println(s);
```

Then, the values are printed in sorted order: `Amy Bob Carl`. As the name of the class suggests, the sorting is accomplished by a tree data structure. (The current implementation uses a *red-black tree*. For a detailed description of red-black trees, see, for example, *Introduction to Algorithms* by Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein [The MIT Press 2001].) Every time an element is added to a tree, it is placed into its proper sorting position. Therefore, the iterator always visits the elements in sorted order.

Adding an element to a tree is slower than adding it to a hash table, but it is still much faster than adding it into the right place in an array or linked list. If the tree contains n elements, then an average of $\log_2 n$ comparisons are required to find the correct position for the new element. For example, if the tree already contains 1,000 elements, then adding a new element requires about 10 comparisons.

Thus, adding elements into a `TreeSet` is somewhat slower than adding into a `HashSet`—see Table 2–3 for a comparison—but the `TreeSet` automatically sorts the elements.

Table 2–3: Adding Elements into Hash and Tree Sets

Document	Total Number of Words	Number of Distinct Words	`HashSet`	`TreeSet`
Alice in Wonderland	28195	5909	5 sec	7 sec
The Count of Monte Cristo	466300	37545	75 sec	98 sec

`java.util.TreeSet<E>` 1.2

- `TreeSet()`
  constructs an empty tree set.
- `TreeSet(Collection<? extends E> elements)`
  constructs a tree set and adds all elements from a collection.

Object Comparison

How does the `TreeSet` know how you want the elements sorted? By default, the tree set assumes that you insert elements that implement the `Comparable` interface. That interface defines a single method:

```
public interface Comparable<T>
{
   int compareTo(T other);
}
```

The call `a.compareTo(b)` must return 0 if `a` and `b` are equal, a negative integer if `a` comes before `b` in the sort order, and a positive integer if `a` comes after `b`. The exact value does not matter; only its sign (>0, 0, or < 0) matters. Several standard Java platform classes implement the `Comparable` interface. One example is the `String` class. Its `compareTo` method compares strings in dictionary order (sometimes called *lexicographic order*).

If you insert your own objects, you must define a sort order yourself by implementing the `Comparable` interface. There is no default implementation of `compareTo` in the `Object` class.

For example, here is how you can sort `Item` objects by part number.

```
class Item implements Comparable<Item>
{
   public int compareTo(Item other)
   {
      return partNumber - other.partNumber;
```

```
      }
      . . .
   }
```

If you compare two *positive* integers, such as part numbers in our example, then you can simply return their difference—it will be negative if the first item should come before the second item, zero if the part numbers are identical, and positive otherwise.

CAUTION: This trick only works if the integers are from a small enough range. If x is a large positive integer and y is a large negative integer, then the difference x – y can overflow.

However, using the Comparable interface for defining the sort order has obvious limitations. A given class can implement the interface only once. But what can you do if you need to sort a bunch of items by part number in one collection and by description in another? Furthermore, what can you do if you need to sort objects of a class whose creator didn't bother to implement the Comparable interface?

In those situations, you tell the tree set to use a different comparison method, by passing a Comparator object into the TreeSet constructor. The Comparator interface declares a compare method with two explicit parameters:

```
public interface Comparator<T>
{
   int compare(T a, T b);
}
```

Just like the compareTo method, the compare method returns a negative integer if a comes before b, zero if they are identical, or a positive integer otherwise.

To sort items by their description, simply define a class that implements the Comparator interface:

```
class ItemComparator implements Comparator<Item>
{
   public int compare(Item a, Item b)
   {
      String descrA = a.getDescription();
      String descrB = b.getDescription();
      return descrA.compareTo(descrB);
   }
}
```

You then pass an object of this class to the tree set constructor:

```
ItemComparator comp = new ItemComparator();
SortedSet<Item> sortByDescription = new TreeSet<Item>(comp);
```

If you construct a tree with a comparator, it uses this object whenever it needs to compare two elements.

Note that this item comparator has no data. It is just a holder for the comparison method. Such an object is sometimes called a *function object*.

Function objects are commonly defined "on the fly," as instances of anonymous inner classes:

```
SortedSet<Item> sortByDescription = new TreeSet<Item>(new
   Comparator<Item>()
   {
      public int compare(Item a, Item b)
      {
```

```
            String descrA = a.getDescription();
            String descrB = b.getDescription();
            return descrA.compareTo(descrB);
         }
      });
```

NOTE: Actually, the `Comparator<T>` interface is declared to have two methods: `compare` and `equals`. Of course, *every* class has an `equals` method; thus, there seems little benefit in adding the method to the interface declaration. The API documentation explains that you need not override the `equals` method but that doing so may yield improved performance in some cases. For example, the `addAll` method of the `TreeSet` class can work more effectively if you add elements from another set that uses the same comparator.

If you look back at Table 2–3, you may well wonder if you should always use a tree set instead of a hash set. After all, adding elements does not seem to take much longer, and the elements are automatically sorted. The answer depends on the data that you are collecting. If you don't need the data sorted, there is no reason to pay for the sorting overhead. More important, with some data it is very difficult to come up with a sort order. Suppose you collect a bunch of rectangles. How do you sort them? By area? You can have two different rectangles with different positions but the same area. If you sort by area, the second one is not inserted into the set. The sort order for a tree must be a *total ordering* Any two elements must be comparable, and the comparison can only be zero if the elements are equal. There is such a sort order for rectangles (the lexicographic ordering on its coordinates), but it is unnatural and cumbersome to compute. In contrast, hash functions are usually easier to define. They only need to do a reasonably good job of scrambling the objects, whereas comparison functions must tell objects apart with complete precision.

The program in Example 2–3 builds two tree sets of `Item` objects. The first one is sorted by part number, the default sort order of `Item` objects. The second set is sorted by description, by means of a custom comparator.

Example 2–3: TreeSetTest.java

```
1. import java.util.*;
2.
3. /**
4.    This program sorts a set of items by comparing
5.    their descriptions.
6. */
7. public class TreeSetTest
8. {
9.    public static void main(String[] args)
10.   {
11.      SortedSet<Item> parts = new TreeSet<Item>();
12.      parts.add(new Item("Toaster", 1234));
13.      parts.add(new Item("Widget", 4562));
14.      parts.add(new Item("Modem", 9912));
15.      System.out.println(parts);
16.
17.      SortedSet<Item> sortByDescription = new TreeSet<Item>(new
18.         Comparator<Item>()
19.         {
20.            public int compare(Item a, Item b)
21.            {
22.               String descrA = a.getDescription();
```

```
23.               String descrB = b.getDescription();
24.               return descrA.compareTo(descrB);
25.            }
26.         });
27.
28.      sortByDescription.addAll(parts);
29.      System.out.println(sortByDescription);
30.   }
31. }
32.
33. /**
34.    An item with a description and a part number.
35. */
36. class Item implements Comparable<Item>
37. {
38.    /**
39.       Constructs an item.
40.       @param aDescription the item's description
41.       @param aPartNumber the item's part number
42.    */
43.    public Item(String aDescription, int aPartNumber)
44.    {
45.       description = aDescription;
46.       partNumber = aPartNumber;
47.    }
48.
49.    /**
50.       Gets the description of this item.
51.       @return the description
52.    */
53.    public String getDescription()
54.    {
55.       return description;
56.    }
57.
58.    public String toString()
59.    {
60.       return "[description=" + description
61.          + ", partNumber=" + partNumber + "]";
62.    }
63.
64.    public boolean equals(Object otherObject)
65.    {
66.       if (this == otherObject) return true;
67.       if (otherObject == null) return false;
68.       if (getClass() != otherObject.getClass()) return false;
69.       Item other = (Item) otherObject;
70.       return description.equals(other.description)
71.          && partNumber == other.partNumber;
72.    }
73.
74.    public int hashCode()
75.    {
76.       return 13 * description.hashCode() + 17 * partNumber;
77.    }
78.
```

```
79.     public int compareTo(Item other)
80.     {
81.        return partNumber - other.partNumber;
82.     }
83.
84.     private String description;
85.     private int partNumber;
86. }
```

java.lang.Comparable<T> 1.2

- `int compareTo(T other)`
  compares this object with another object and returns a negative value if `this` comes before `other`, zero if they are considered identical in the sort order, and a positive value if `this` comes after `other`.

java.util.Comparator<T> 1.2

- `int compare(T a, T b)`
  compares two objects and returns a negative value if `a` comes before `b`, zero if they are considered identical in the sort order, and a positive value if `a` comes after `b`.

java.util.SortedSet<E> 1.2

- `Comparator<? super E> comparator()`
  returns the comparator used for sorting the elements, or `null` if the elements are compared with the `compareTo` method of the `Comparable` interface.
- `E first()`
- `E last()`
  return the smallest or largest element in the sorted set.

java.util.TreeSet<E> 1.2

- `TreeSet()`
  constructs a tree set for storing `Comparable` objects.
- `TreeSet(Comparator<? super E> c)`
  constructs a tree set and uses the specified comparator for sorting its elements.
- `TreeSet(SortedSet<? extends E> elements)`
  constructs a tree set, adds all elements from a sorted set, and uses the same element comparator as the given sorted set.

Priority Queues

A priority queue retrieves elements in sorted order after they were inserted in arbitrary order. That is, whenever you call the `remove` method, you get the smallest element currently in the priority queue. However, the priority queue does not sort all its elements. If you iterate over the elements, they are not necessarily sorted. The priority queue makes use of an elegant and efficient data structure, called a *heap*. A heap is a self-organizing binary tree in which the `add` and `remove` operations cause the smallest element to gravitate to the root, without wasting time on sorting all elements.

Just like a `TreeSet`, a priority queue can either hold elements of a class that implements the `Comparable` interface or a `Comparator` object you supply in the constructor.

A typical use for a priority queue is job scheduling. Each job has a priority. Jobs are added in random order. Whenever a new job can be started, the highest-priority job is removed

from the queue. (Since it is traditional for priority 1 to be the "highest" priority, the `remove` operation yields the minimum element.)

Example 2–4 shows a priority queue in action. Unlike iteration in a `TreeSet`, the iteration here does not visit the elements in sorted order. However, removal always yields the smallest remaining element.

Example 2–4: PriorityQueueTest.java

```
1. import java.util.*;
2.
3. /**
4.    This program demonstrates the use of a priority queue.
5. */
6. public class PriorityQueueTest
7. {
8.    public static void main(String[] args)
9.    {
10.      PriorityQueue<GregorianCalendar> pq = new PriorityQueue<GregorianCalendar>();
11.      pq.add(new GregorianCalendar(1906, Calendar.DECEMBER, 9)); // G. Hopper
12.      pq.add(new GregorianCalendar(1815, Calendar.DECEMBER, 10)); // A. Lovelace
13.      pq.add(new GregorianCalendar(1903, Calendar.DECEMBER, 3)); // J. von Neumann
14.      pq.add(new GregorianCalendar(1910, Calendar.JUNE, 22)); // K. Zuse
15.
16.      System.out.println("Iterating over elements...");
17.      for (GregorianCalendar date : pq)
18.         System.out.println(date.get(Calendar.YEAR));
19.      System.out.println("Removing elements...");
20.      while (!pq.isEmpty())
21.         System.out.println(pq.remove().get(Calendar.YEAR));
22.   }
23. }
```

`java.util.PriorityQueue` 5.0

- `PriorityQueue()`
- `PriorityQueue(int initialCapacity)`
  construct a tree set for storing `Comparable` objects.
- `PriorityQueue(int initialCapacity, Comparator<? super E> c)`
  constructs a tree set and uses the specified comparator for sorting its elements.

Maps

A set is a collection that lets you quickly find an existing element. However, to look up an element, you need to have an exact copy of the element to find. That isn't a very common lookup—usually, you have some key information, and you want to look up the associated element. The *map* data structure serves that purpose. A map stores key/value pairs. You can find a value if you provide the key. For example, you may store a table of employee records, where the keys are the employee IDs and the values are `Employee` objects.

The Java library supplies two general-purpose implementations for maps: `HashMap` and `TreeMap`. Both classes implement the `Map` interface.

A hash map hashes the keys, and a tree map uses a total ordering on the keys to organize them in a search tree. The hash or comparison function is applied *only to the keys*. The values associated with the keys are not hashed or compared.

Should you choose a hash map or a tree map? As with sets, hashing is a bit faster, and it is the preferred choice if you don't need to visit the keys in sorted order.

Here is how you set up a hash map for storing employees.

```
Map<String, Employee> staff = new HashMap<String, Employee>(); // HashMap implements Map
Employee harry = new Employee("Harry Hacker");
staff.put("987-98-9996", harry);
. . .
```

Whenever you add an object to a map, you must supply a key as well. In our case, the key is a string, and the corresponding value is an `Employee` object.

To retrieve an object, you must use (and, therefore, remember) the key.

```
String s = "987-98-9996";
e = staff.get(s); // gets harry
```

If no information is stored in the map with the particular key specified, then `get` returns `null`.

Keys must be unique. You cannot store two values with the same key. If you call the `put` method twice with the same key, then the second value replaces the first one. In fact, `put` returns the previous value stored with the key parameter.

The `remove` method removes an element with a given key from the map. The `size` method returns the number of entries in the map.

The collections framework does not consider a map itself as a collection. (Other frameworks for data structures consider a map as a collection of *pairs*, or as a collection of values that is indexed by the keys.) However, you can obtain *views* of the map, objects that implement the `Collection` interface, or one of its subinterfaces.

There are three views: the set of keys, the collection of values (which is not a set), and the set of key/value pairs. The keys and key/value pairs form a set because there can be only one copy of a key in a map. The methods

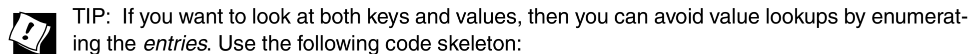
```
Set<K> keySet()
Collection<K> values()
Set<Map.Entry<K, V>> entrySet()
```

return these three views. (The elements of the entry set are objects of the static inner class `Map.Entry`.)

Note that the `keySet` is *not* a `HashSet` or `TreeSet`, but an object of some other class that implements the `Set` interface. The `Set` interface extends the `Collection` interface. Therefore, you can use a `keySet` as you would use any collection.

For example, you can enumerate all keys of a map:

```
Set<String> keys = map.keySet();
for (String key : keys)
{
   do something with key
}
```



TIP: If you want to look at both keys and values, then you can avoid value lookups by enumerating the *entries*. Use the following code skeleton:

```
for (Map.Entry<String, Employee> entry : staff.entrySet())
{
   String key = entry.getKey();
   Employee value = entry.getValue();
   do something with key, value
}
```

If you invoke the `remove` method of the iterator, you actually remove the key *and its associated value* from the map. However, you cannot *add* an element to the key set view. It makes no sense to add a key without also adding a value. If you try to invoke the `add` method, it throws an `UnsupportedOperationException`. The entry set view has the same restriction, even though it would make conceptual sense to add a new key/value pair.

Example 2–5 illustrates a map at work. We first add key/value pairs to a map. Then, we remove one key from the map, which removes its associated value as well. Next, we change the value that is associated with a key and call the `get` method to look up a value. Finally, we iterate through the entry set.

Example 2–5: MapTest.java

```
1. import java.util.*;
2.
3. /**
4.    This program demonstrates the use of a map with key type
5.    String and value type Employee.
6. */
7. public class MapTest
8. {
9.    public static void main(String[] args)
10.    {
11.       Map<String, Employee> staff = new HashMap<String, Employee>();
12.       staff.put("144-25-5464", new Employee("Amy Lee"));
13.       staff.put("567-24-2546", new Employee("Harry Hacker"));
14.       staff.put("157-62-7935", new Employee("Gary Cooper"));
15.       staff.put("456-62-5527", new Employee("Francesca Cruz"));
16.
17.       // print all entries
18.
19.       System.out.println(staff);
20.
21.       // remove an entry
22.
23.       staff.remove("567-24-2546");
24.
25.       // replace an entry
26.
27.       staff.put("456-62-5527", new Employee("Francesca Miller"));
28.
29.       // look up a value
30.
31.       System.out.println(staff.get("157-62-7935"));
32.
33.       // iterate through all entries
34.
35.       for (Map.Entry<String, Employee> entry : staff.entrySet())
36.       {
37.          String key = entry.getKey();
38.          Employee value = entry.getValue();
39.          System.out.println("key=" + key + ", value=" + value);
40.       }
41.    }
42. }
43.
```

```
44. /**
45.    A minimalist employee class for testing purposes.
46. */
47. class Employee
48. {
49.    /**
50.       Constructs an employee with $0 salary.
51.       @param n the employee name
52.    */
53.    public Employee(String n)
54.    {
55.       name = n;
56.       salary = 0;
57.    }
58.
59.    public String toString()
60.    {
61.       return "[name=" + name + ", salary=" + salary + "]";
62.    }
63.
64.    private String name;
65.    private double salary;
66. }
```

java.util.Map<K, V> 1.2

- `V get(K key)`
  gets the value associated with the key; returns the object associated with the key, or `null` if the key is not found in the map. The key may be `null`.
- `V put(K key, V value)`
  puts the association of a key and a value into the map. If the key is already present, the new object replaces the old one previously associated with the key. This method returns the old value of the key, or `null` if the key was not previously present. The key may be `null`, but the value must not be `null`.
- `void putAll(Map<? extends K, ? extends V> entries)`
  adds all entries from the specified map to this map.
- `boolean containsKey(Object key)`
  returns `true` if the key is present in the map.
- `boolean containsValue(Object value)`
  returns `true` if the value is present in the map.
- `Set<Map.Entry<K, V>> entrySet()`
  returns a set view of `Map.Entry` objects, the key/value pairs in the map. You can remove elements from this set and they are removed from the map, but you cannot add any elements.
- `Set<K> keySet()`
  returns a set view of all keys in the map. You can remove elements from this set and the keys and associated values are removed from the map, but you cannot add any elements.
- `Collection<V> values()`
  returns a collection view of all values in the map. You can remove elements from this set and the removed value and its key are removed from the map, but you cannot add any elements.

java.util.Map.Entry<K, V> 1.2

- `K getKey()`
- `V getValue()`
  return the key or value of this entry.
- `V setValue(V newValue)`
  changes the value *in the associated map* to the new value and returns the old value.

java.util.HashMap<K, V> 1.2

- `HashMap()`
- `HashMap(int initialCapacity)`
- `HashMap(int initialCapacity, float loadFactor)`
  construct an empty hash map with the specified capacity and load factor (a number between 0.0 and 1.0 that determines at what percentage of fullness the hash table will be rehashed into a larger one). The default load factor is 0.75.

java.util.TreeMap<K,V> 1.2

- `TreeMap(Comparator<? super K> c)`
  constructs a tree map and uses the specified comparator for sorting its keys.
- `TreeMap(Map<? extends K, ? extends V> entries)`
  constructs a tree map and adds all entries from a map.
- `TreeMap(SortedMap<? extends K, ? extends V> entries)`
  constructs a tree map, adds all entries from a sorted map, and uses the same element comparator as the given sorted map.

***java.util.SortedMap<K, V>* 1.2**

- `Comparator<? super K> comparator()`
  returns the comparator used for sorting the keys, or `null` if the keys are compared with the `compareTo` method of the `Comparable` interface.
- `K firstKey()`
- `K lastKey()`
  return the smallest or largest key in the map.

Specialized Set and Map Classes

The collection class library has several map classes for specialized needs that we briefly discuss in this section.

Weak Hash Maps

The `WeakHashMap` class was designed to solve an interesting problem. What happens with a value whose key is no longer used anywhere in your program? Suppose the last reference to a key has gone away. Then, there is no longer any way to refer to the value object. But because no part of the program has the key any more, the key/value pair cannot be removed from the map. Why can't the garbage collector remove it? Isn't it the job of the garbage collector to remove unused objects?

Unfortunately, it isn't quite so simple. The garbage collector traces *live* objects. As long as the map object is live, then *all* buckets in it are live and they won't be reclaimed. Thus, your program should take care to remove unused values from long-lived maps. Or, you can use a `WeakHashMap` instead. This data structure cooperates with the garbage collector to remove key/value pairs when the only reference to the key is the one from the hash table entry.

Here are the inner workings of this mechanism. The `WeakHashMap` uses *weak references* to hold keys. A `WeakReference` object holds a reference to another object, in our case, a hash table key. Objects of this type are treated in a special way by the garbage collector. Normally, if the garbage collector finds that a particular object has no references to it, it simply reclaims the object. However, if the object is reachable *only* by a `WeakReference`, the garbage collector still reclaims the object, but it places the weak reference that led to it into a queue. The operations of the `WeakHashMap` periodically check that queue for newly arrived weak references. The arrival of a weak reference in the queue signifies that the key was no longer used by anyone and that it has been collected. The `WeakHashMap` then removes the associated entry.

Linked Hash Sets and Maps

JDK 1.4 adds classes `LinkedHashSet` and `LinkedHashMap` that remember in which order you inserted items. That way, you avoid the seemingly random order of items in a hash table. As entries are inserted into the table, they are joined in a doubly linked list (see Figure 2–9).

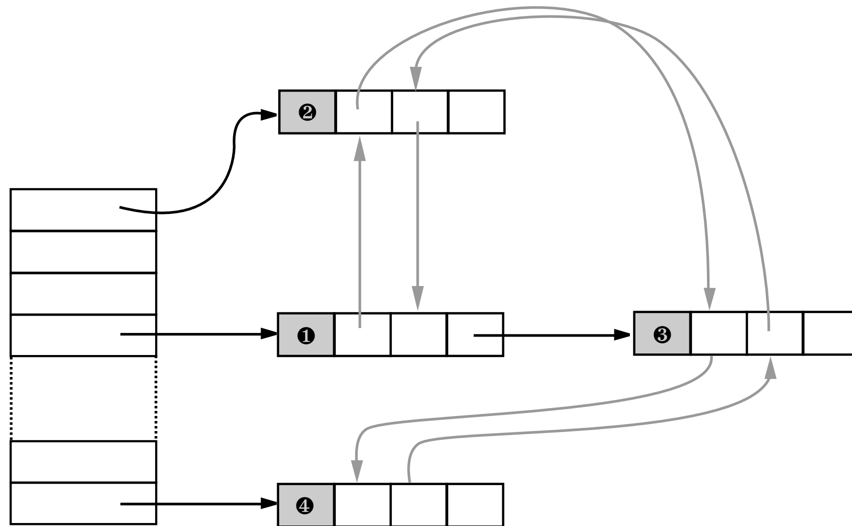


Figure 2–9: A linked hash table

For example, consider the following map insertions from Example 2–5:

```
Map staff = new LinkedHashMap();
staff.put("144-25-5464", new Employee("Amy Lee"));
staff.put("567-24-2546", new Employee("Harry Hacker"));
staff.put("157-62-7935", new Employee("Gary Cooper"));
staff.put("456-62-5527", new Employee("Francesca Cruz"));
```

Then `staff.keySet().iterator()` enumerates the keys in the order:

```
144-25-5464
567-24-2546
157-62-7935
456-62-5527
```

and `staff.values().iterator()` enumerates the values in the order:

```
Amy Lee
Harry Hacker
Gary Cooper
Francesca Cruz
```

CJ7V2.book Page 116 Wednesday, October 27, 2004 2:32 PM

A linked hash map can alternatively use *access order*, not insertion order, to iterate through the map entries. Every time you call `get` or `put`, the affected entry is removed from its current position and placed at the *end* of the linked list of entries. (Only the position in the linked list of entries is affected, not the hash table bucket. An entry always stays in the bucket that corresponds to the hash code of the key.) To construct such a hash map, call

```
LinkedHashMap<K, V>(initialCapacity, loadFactor, true)
```

Access order is useful for implementing a "least recently used" discipline for a cache. For example, you may want to keep frequently accessed entries in memory and read less frequently accessed objects from a database. When you don't find an entry in the table, and the table is already pretty full, then you can get an iterator into the table and remove the first few elements that it enumerates. Those entries were the least recently used ones.

You can even automate that process. Form a subclass of `LinkedHashMap` and override the method

```
protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
```

Adding a new entry then causes the `eldest` entry to be removed whenever your method returns `true`. For example, the following cache is kept at a size of at most 100 elements:

```
Map<K, V> cache = new
   LinkedHashMap<K, V>(128, 0.75F, true)
   {
      protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
      {
         return size() > 100;
      }
   };
```

Alternatively, you can consider the `eldest` entry to decide whether to remove it. For example, you may want to check a time stamp stored with the entry.

Enumeration Sets and Maps

The `EnumSet` is an efficient set implementation with elements that belong to an enumerated type. Because an enumerated type has a finite number of instances, the `EnumSet` is internally implemented simply as a sequence of bits. A bit is turned on if the corresponding value is present in the set.

The `EnumSet` class has no public constructors. You use a static factory method to construct the set:

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };
EnumSet<Weekday> always = EnumSet.allOf(Weekday.class);
EnumSet<Weekday> never = EnumSet.noneOf(Weekday.class);
EnumSet<Weekday> workday = EnumSet.range(Weekday.MONDAY, Weekday.FRIDAY);
EnumSet<Weekday> mwf = EnumSet.of(Weekday.MONDAY, Weekday.WEDNESDAY, Weekday.FRIDAY);
```

You can use the usual methods of the `Set` interface to modify an `EnumSet`.

An `EnumMap` is a map with keys that belong to an enumerated type. It is simply and efficiently implemented as an array of values. You need to specify the key type in the constructor:

```
EnumMap<Weekday, Employee> personInCharge = new EnumMap<Weekday, Employee>(Weekday.class);
```

NOTE: In the API documentation for `EnumSet`, you will see odd-looking type parameters of the form `E extends Enum<E>`. This simply means "`E` is an enumerated type." All enumerated types extend the generic `Enum` class. For example, `Weekday` extends `Enum<Weekday>`.

CJ7V2.book Page 117 Wednesday, October 27, 2004 2:32 PM

Identity Hash Maps

JDK 1.4 adds another class IdentityHashMap for another quite specialized purpose, where the hash values for the keys should not be computed by the hashCode method but by the System.identityHashCode method. That's the method that Object.hashCode uses to compute a hash code from the object's memory address. Also, for comparison of objects, the IdentityHashMap uses ==, not equals.

In other words, different key objects are considered distinct even if they have equal contents. This class is useful for implementing object traversal algorithms (such as object serialization), in which you want to keep track of which objects have already been traversed.

java.util.WeakHashMap<K, V> 1.2

- WeakHashMap()
- WeakHashMap(int initialCapacity)
- WeakHashMap(int initialCapacity, float loadFactor)
  construct an empty hash map with the specified capacity and load factor.

java.util.LinkedHashSet<E> 1.4

- LinkedHashSet()
- LinkedHashSet(int initialCapacity)
- LinkedHashSet(int initialCapacity, float loadFactor)
  construct an empty linked hash set with the specified capacity and load factor.

java.util.LinkedHashMap<K, V> 1.4

- LinkedHashMap()
- LinkedHashMap(int initialCapacity)
- LinkedHashMap(int initialCapacity, float loadFactor)
- LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)
  construct an empty linked hash map with the specified capacity, load factor, and ordering. The accessOrder parameter is true for access order, false for insertion order.
- protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
  should be overridden to return true if you want the eldest entry to be removed. The eldest parameter is the entry whose removal is being contemplated. This method is called after an entry has been added to the map. The default implementation returns false—old elements are not removed by default. However, you can redefine this method to selectively return true; for example, if the eldest entry fits a certain condition or the map exceeds a certain size.

java.util.EnumSet<E extends Enum<E>> 5.0

- static <E extends Enum<E>> EnumSet<E> allOf(Class<E> enumType)
  returns a set that contains all values of the given enumerated type.
- static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> enumType)
  returns an empty set, capable of holding values of the given enumerated type.
- static <E extends Enum<E>> EnumSet<E> range(E from, E to)
  returns a set that contains all values between from and to (inclusive).
- static <E extends Enum<E>> EnumSet<E> of(E value)
- static <E extends Enum<E>> EnumSet<E> of(E value, E... values)
  return a set that contains the given values.

java.util.EnumMap<K extends Enum<K>, V> 5.0

- `EnumMap(Class<K> keyType)`
  constructs an empty map whose keys have the given type.

java.util.IdentityHashMap<K, V> 1.4

- `IdentityHashMap()`
- `IdentityHashMap(int expectedMaxSize)`
  construct an empty identity hash map whose capacity is the smallest power of 2 exceeding 1.5 * `expectedMaxSize`. (The default for `expectedMaxSize` is 21.)

java.lang.System 1.0

- `static int identityHashCode(Object obj)` **1.1**
  returns the same hash code (derived from the object's memory address) that `Object.hashCode` computes, even if the class to which `obj` belongs has redefined the `hashCode` method.

The Collections Framework

A *framework* is a set of classes that form the basis for building advanced functionality. A framework contains superclasses with useful functionality, policies, and mechanisms. The user of a framework forms subclasses to extend the functionality without having to reinvent the basic mechanisms. For example, Swing is a framework for user interfaces.

The Java collections library forms a framework for collection classes. It defines a number of interfaces and abstract classes for implementors of collections (see Figure 2–10), and it prescribes certain mechanisms, such as the iteration protocol. You can use the collection classes without having to know much about the framework—we did just that in the preceding sections. However, if you want to implement generic algorithms that work for multiple collection types or if you want to add a new collection type, it is helpful to understand the framework.

There are two fundamental interfaces for collections: `Collection` and `Map`. You insert elements into a collection with a method:

```
boolean add(E element)
```

However, maps hold key/value pairs, and you use the `put` method to insert them.

```
V put(K key, V value)
```

To read elements from a collection, you visit them with an iterator. However, you can read values from a map with the `get` method:

```
V get(K key)
```

A `List` is an *ordered collection*. Elements are added into a particular position in the container. An object can be placed into its position in two ways: by an integer index and by a list iterator. The `List` interface defines methods for random access:

```
void add(int index, E element)
E get(int index)
void remove(int index)
```

As already discussed, the `List` interface provides these random access methods whether or not they are efficient for a particular implementation. To make it possible to avoid carrying out costly random access operations, JDK 1.4 introduces a tagging interface, `RandomAccess`. That interface has no methods, but you can use it to test whether a particular collection supports efficient random access:

```
if (c instanceof RandomAccess)
{
   use random access algorithm
}
else
{
   use sequential access algorithm
}
```

The ArrayList and Vector classes implement the RandomAccess interface.

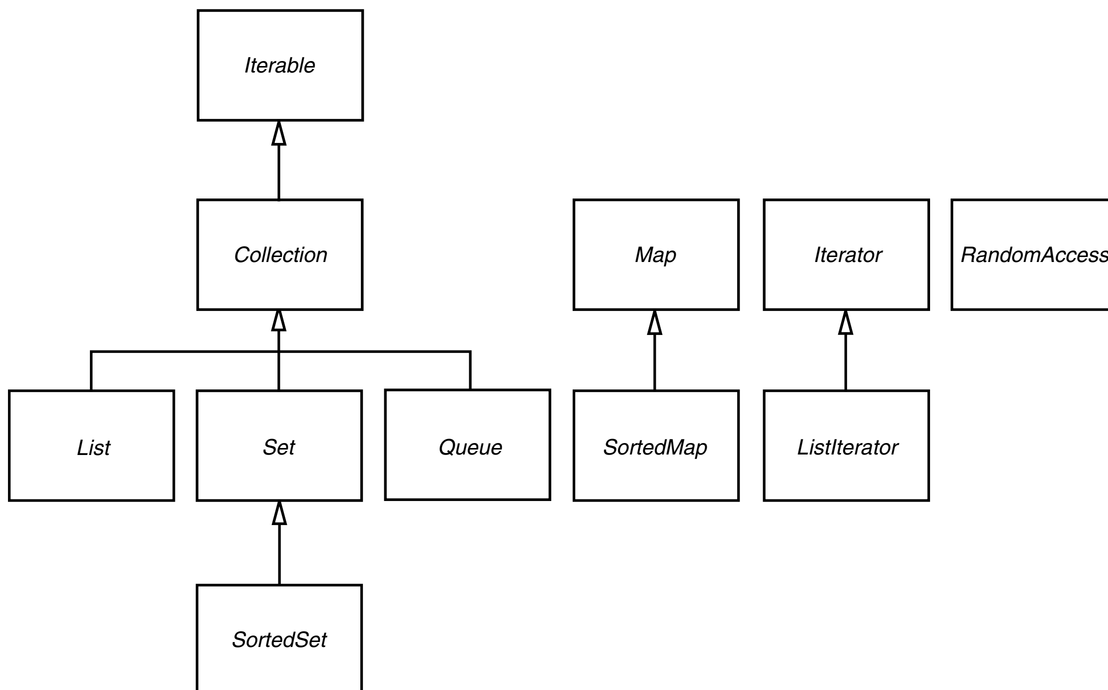


Figure 2–10: The interfaces of the collections framework

NOTE: From a theoretical point of view, it would have made sense to have a separate Array interface that extends the List interface and declares the random access methods. If there were a separate Array interface, then those algorithms that require random access would use Array parameters and you could not accidentally apply them to collections with slow random access. However, the designers of the collections framework chose not to define a separate interface, because they wanted to keep the number of interfaces in the library small. Also, they did not want to take a paternalistic attitude toward programmers. You are free to pass a linked list to algorithms that use random access—you just need to be aware of the performance costs.

The ListIterator interface defines a method for adding an element before the iterator position:

```
void add(E element)
```

To get and remove elements at a particular position, you simply use the next and remove methods of the Iterator interface.

The Set interface is identical to the Collection interface, but the behavior of the methods is more tightly defined. The add method of a set should reject duplicates. The equals method of a set should be defined so that two sets are identical if they have the same elements, but not necessarily in the same order. The hashCode method should be defined such that two sets with the same elements yield the same hash code.

Why make a separate interface if the method signatures are the same? Conceptually, not all collections are sets. Making a Set interface enables programmers to write methods that accept only sets.

Finally, the SortedSet and SortedMap interfaces expose the comparison object used for sorting, and they define methods to obtain views of subsets of the collections. We discuss these views in the next section.

Now, let us turn from the interfaces to the classes that implement them. We already discussed that the collection interfaces have quite a few methods that can be trivially implemented from more fundamental methods. Abstract classes supply many of these routine implementations:

```
AbstractCollection
AbstractList
AbstractSequentialList
AbstractSet
AbstractQueue
AbstractMap
```

If you implement your own collection class, then you probably want to extend one of these classes so that you can pick up the implementations of the routine operations.

The Java library supplies concrete classes:

```
LinkedList
ArrayList
HashSet
TreeSet
PriorityQueue
HashMap
TreeMap
```

Figure 2–11 shows the relationships between these classes.

Finally, a number of “legacy” container classes have been present since JDK 1.0, before there was a collections framework:

```
Vector
Stack
Hashtable
Properties
```

They have been integrated into the collections framework—see Figure 2–12. We discuss these classes later in this chapter.

Views and Wrappers

If you look at Figure 2–10 and Figure 2–11, you might think it is overkill to have lots of interfaces and abstract classes to implement a modest number of concrete collection classes. However, these figures don’t tell the whole story. By using *views*, you can obtain other objects that implement the Collection or Map interfaces. You saw one example of this with the keySet method of the map classes. At first glance, it appears as if the method creates a new set, fills it with all keys of the map, and returns it. However, that is not the case. Instead, the keySet method returns an object of a class that implements the Set interface and whose methods manipulate the original map. Such a collection is called a *view*.

CJ7V2.book Page 121 Wednesday, October 27, 2004 2:32 PM

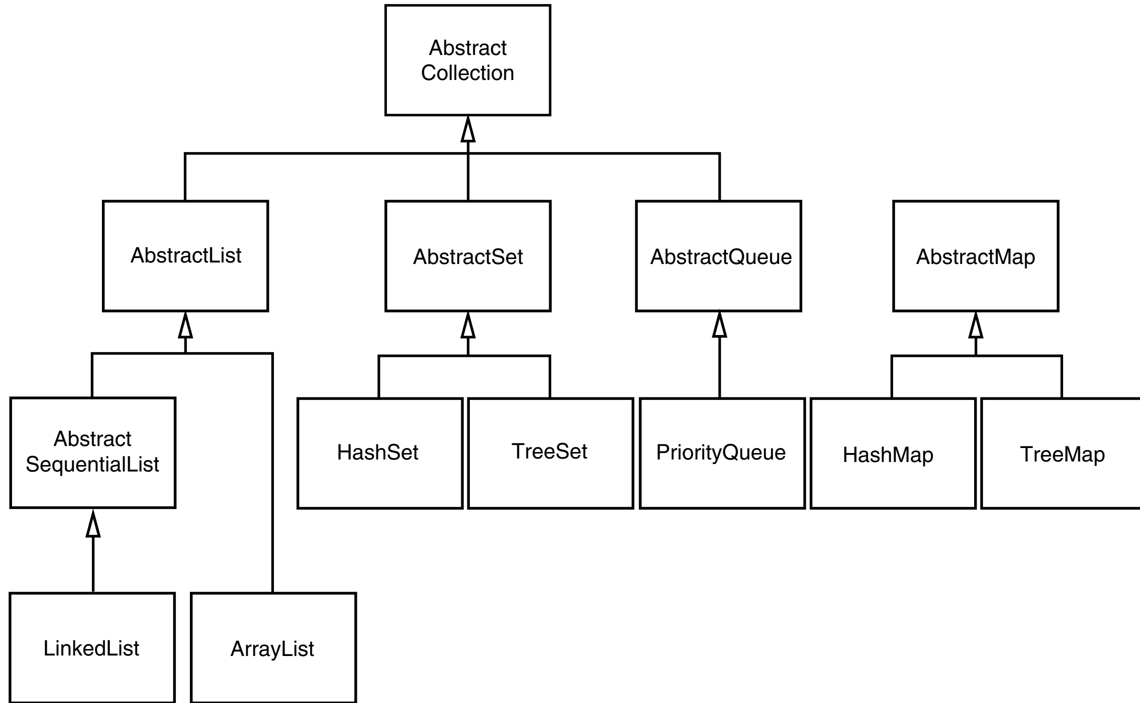


Figure 2–11: Classes in the collections framework

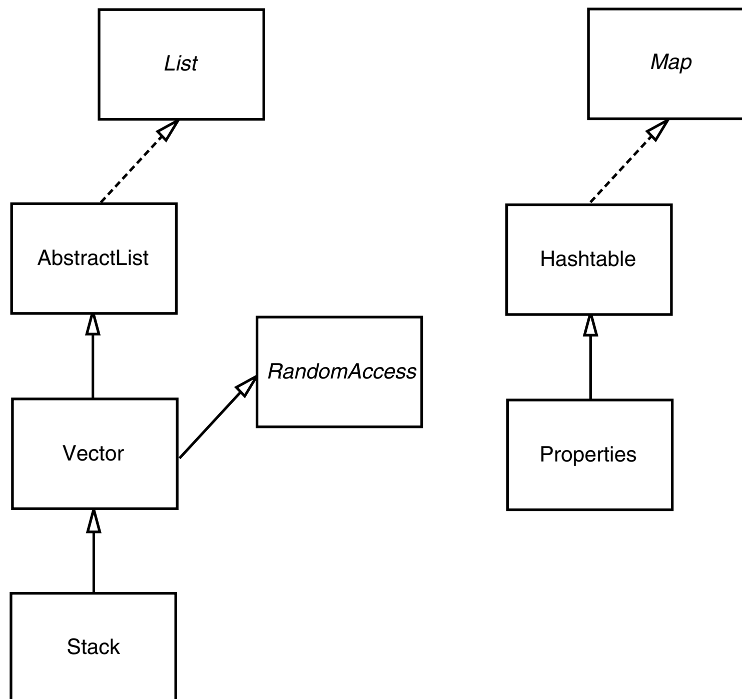


Figure 2–12: Legacy classes in the collections framework

The technique of views has a number of useful applications in the collections framework. We discuss these applications in the following sections.

Lightweight Collection Wrappers

The static `asList` method of the `Arrays` class returns a `List` wrapper around a plain Java array. This method lets you pass the array to a method that expects a list or collection argument. For example,

```
Card[] cardDeck = new Card[52];
. . .
List<Card> cardList = Arrays.asList(cardDeck);
```

The returned object is *not* an `ArrayList`. It is a view object with `get` and `set` methods that access the underlying array. All methods that would change the size of the array (such as `add` and the `remove` method of the associated iterator) throw an `UnsupportedOperationException`.

As of JDK 5.0, the `asList` method is declared to have a variable number of arguments. Instead of passing an array, you can also pass individual elements. For example,

```
List<String> names = Arrays.asList("Amy", "Bob", "Carl");
```

The method call

```
Collections.nCopies(n, anObject)
```

returns an immutable object that implements the `List` interface and gives the illusion of having `n` elements, each of which appears as `anObject`.

For example, the following call creates a `List` containing 100 strings, all set to `"DEFAULT"`:

```
List<String> settings = Collections.nCopies(100, "DEFAULT");
```

There is very little storage cost—the object is stored only once. This is a cute application of the view technique.

NOTE: The `Collections` class contains a number of utility methods with parameters or return values that are collections. Do not confuse it with the `Collection` interface.

The method call

```
Collections.singleton(anObject)
```

returns a view object that implements the `Set` interface (unlike `ncopies`, which produces a `List`). The returned object implements an immutable single-element set without the overhead of data structure. The methods `singletonList` and `singletonMap` behave similarly.

Subranges

You can form subrange views for a number of collections. For example, suppose you have a list `staff` and want to extract elements 10 to 19. You use the `subList` method to obtain a view into the subrange of the list.

```
List group2 = staff.subList(10, 20);
```

The first index is inclusive, the second exclusive—just like the parameters for the `substring` operation of the `String` class.

You can apply any operations to the subrange, and they automatically reflect the entire list. For example, you can erase the entire subrange:

```
group2.clear(); // staff reduction
```

The elements are now automatically cleared from the `staff` list, and `group2` is empty.

For sorted sets and maps, you use the sort order, not the element position, to form subranges. The `SortedSet` interface declares three methods:

CJ7V2.book Page 123 Wednesday, October 27, 2004 2:32 PM

```
subSet(from, to)
headSet(to)
tailSet(from)
```

These return the subsets of all elements that are larger than or equal to `from` and strictly smaller than `to`. For sorted maps, the similar methods

```
subMap(from, to)
headMap(to)
tailMap(from)
```

return views into the maps consisting of all entries in which the *keys* fall into the specified ranges.

Unmodifiable Views

The `Collections` class has methods that produce *unmodifiable views* of collections. These views add a runtime check to an existing collection. If an attempt to modify the collection is detected, then an exception is thrown and the collection remains untouched.

You obtain unmodifiable views by six methods:

```
Collections.unmodifiableCollection
Collections.unmodifiableList
Collections.unmodifiableSet
Collections.unmodifiableSortedSet
Collections.unmodifiableMap
Collections.unmodifiableSortedMap
```

Each method is defined to work on an interface. For example, `Collections.unmodifiableList` works with an `ArrayList`, a `LinkedList`, or any other class that implements the `List` interface.

For example, suppose you want to let some part of your code look at, but not touch, the contents of a collection. Here is what you could do:

```
List<String> staff = new LinkedList<String>();
. . .
lookAt(new Collections.unmodifiableList(staff));
```

The `Collections.unmodifiableList` method returns an object of a class implementing the `List` interface. Its accessor methods retrieve values from the `staff` collection. Of course, the `lookAt` method can call all methods of the `List` interface, not just the accessors. But all mutator methods (such as `add`) have been redefined to throw an `UnsupportedOperationException` instead of forwarding the call to the underlying collection.

The unmodifiable view does not make the collection itself immutable. You can still modify the collection through its original reference (`staff`, in our case). And you can still call mutator methods on the elements of the collection.

Because the views wrap the *interface* and not the actual collection object, you only have access to those methods that are defined in the interface. For example, the `LinkedList` class has convenience methods, `addFirst` and `addLast`, that are not part of the `List` interface. These methods are not accessible through the unmodifiable view.

CAUTION: The `unmodifiableCollection` method (as well as the `synchronizedCollection` and `checkedCollection` methods discussed later in this section) returns a collection whose `equals` method does *not* invoke the `equals` method of the underlying collection. Instead, it inherits the `equals` method of the `Object` class, which just tests whether the objects are identical. If you turn a set or list into just a collection, you can no longer test for equal contents. The view acts in this way because equality testing is not well defined at this level of the hierarchy. The views treat the `hashCode` method in the same way.

However, the `unmodifiableSet` and `unmodifiableList` class do not hide the `equals` and `hashCode` methods of the underlying collections.

Synchronized Views

If you access a collection from multiple threads, you need to ensure that the collection is not accidentally damaged. For example, it would be disastrous if one thread tried to add to a hash table while another thread was rehashing the elements.

Instead of implementing thread-safe collection classes, the library designers used the view mechanism to make regular collections thread safe. For example, the static `synchronizedMap` method in the `Collections` class can turn any map into a `Map` with synchronized access methods:

```
HashMap<String, Employee> hashMap = new HashMap<String, Employee>();
Map<String, Employee> map = Collections.synchronizedMap(hashMap);
```

You can now access the `map` object from multiple threads. The methods such as `get` and `put` are serialized—each method call must be finished completely before another thread can call another method.

You should make sure that no thread accesses the data structure through the original unsynchronized methods. The easiest way to ensure this is not to save any reference to the original object:

```
map = Collections.synchronizedMap(new HashMap<String, Employee>());
```

Note that the views only serialize the methods of the *collection*. If you use an iterator, you need to manually acquire the lock on the collection object. For example,

```
synchronized (map)
{
   Iterator<String> iter = map.keySet().iterator();
   while (iter.hasNext()) . . .
}
```

You must use the same code if you use a “for each” loop since the loop uses an iterator. Note that the iterator will actually fail with a `ConcurrentModificationException` if another thread modifies the collection while the iteration is in progress. The synchronization is still required so that the concurrent modification can be reliably detected.

As a practical matter, the synchronization wrappers have limited utility. You are usually better off using the collections defined in the `java.util.concurrent` package—see Chapter 1 for more information. In particular, the `ConcurrentHashMap` map has been carefully implemented so that multiple threads can access it without blocking each other, provided they access different buckets.

Checked Views

JDK 5.0 adds a set of “checked” views that are intended as debugging support for a problem that can occur with generic types. As explained in Volume 1, Chapter 13, it is actually possible to smuggle elements of the wrong type into a generic collection. For example,

```
ArrayList<String> strings = new ArrayList<String>();
ArrayList rawList = strings; // get warning only, not an error, for compatibility with legacy code
rawList.add(new Date()); // now strings contains a Date object!
```

The erroneous `add` command is not detected at run time. Instead, a class cast exception will happen later when another part of the code calls `get` and casts the result to a `String`.

A checked view can detect this problem. Define

```
List<String> safeStrings = Collections.checkedList(strings, String.class);
```

CJ7V2.book Page 125 Wednesday, October 27, 2004 2:32 PM

The view's `add` method checks that the inserted object belongs to the given class and immediately throws a `ClassCastException` if it does not. The advantage is that the error is reported at the correct location:

```
ArrayList rawList = safeStrings;
rawList.add(new Date()); // Checked list throws a ClassCastException
```

CAUTION: The checked views are limited by the runtime checks that the virtual machine can carry out. For example, if you have an `ArrayList<Pair<String>>`, you cannot protect it from inserting a `Pair<Date>` since the virtual machine has a single "raw" `Pair` class.

A Note on Optional Operations

A view usually has some restriction—it may be read-only, it may not be able to change the size, or it may support removal, but not insertion, as is the case for the key view of a map. A restricted view throws an `UnsupportedOperationException` if you attempt an inappropriate operation.

In the API documentation for the collection and iterator interfaces, many methods are described as "optional operations." This seems to be in conflict with the notion of an interface. After all, isn't the purpose of an interface to lay out the methods that a class *must* implement? Indeed, this arrangement is unsatisfactory from a theoretical perspective. A better solution might have been to design separate interfaces for read-only views and views that can't change the size of a collection. However, that would have tripled the number of interfaces, which the designers of the library found unacceptable.

Should you extend the technique of "optional" methods to your own designs? We think not. Even though collections are used frequently, the coding style for implementing them is not typical for other problem domains. The designers of a collection class library have to resolve a particularly brutal set of conflicting requirements. Users want the library to be easy to learn, convenient to use, completely generic, idiot proof, and at the same time as efficient as hand-coded algorithms. It is plainly impossible to achieve all these goals simultaneously, or even to come close. But in your own programming problems, you will rarely encounter such an extreme set of constraints. You should be able to find solutions that do not rely on the extreme measure of "optional" interface operations.

`java.util.Collections` 1.2

- `static <E> Collection unmodifiableCollection(Collection<E> c)`
- `static <E> List unmodifiableList(List<E> c)`
- `static <E> Set unmodifiableSet(Set<E> c)`
- `static <E> SortedSet unmodifiableSortedSet(SortedSet<E> c)`
- `static <K, V> Map unmodifiableMap(Map<K, V> c)`
- `static <K, V> SortedMap unmodifiableSortedMap(SortedMap<K, V> c)`

  construct a view of the collection whose mutator methods throw an `UnsupportedOperationException`.

- `static <E> Collection<E> synchronizedCollection(Collection<E> c)`
- `static <E> List synchronizedList(List<E> c)`
- `static <E> Set synchronizedSet(Set<E> c)`
- `static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)`
- `static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)`
- `static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)`

  construct a view of the collection whose methods are synchronized.

CJ7V2.book Page 126 Wednesday, October 27, 2004 2:32 PM

- `static <E> Collection checkedCollection(Collection<E> c, Class<E> elementType)`
- `static <E> List checkedList(List<E> c, Class<E> elementType)`
- `static <E> Set checkedSet(Set<E> c, Class<E> elementType)`
- `static <E> SortedSet checkedSortedSet(SortedSet<E> c, Class<E> elementType)`
- `static <K, V> Map checkedMap(Map<K, V> c, Class<K> keyType, Class<V> valueType)`
- `static <K, V> SortedMap checkedSortedMap(SortedMap<K, V> c, Class<K> keyType, Class<V> valueType)`

  construct a view of the collection whose methods throw a `ClassCastException` if an element of the wrong type is inserted.

- `static <E> List<E> nCopies(int n, E value)`
- `static <E> Set<E> singleton(E value)`

  construct a view of the object as either an unmodifiable list with `n` identical elements, or a set with a single element.

java.util.Arrays 1.2

- `static <E> List<E> asList(E... array)`

  returns a list view of the elements in an array that is modifiable but not resizable.

***java.util.List<E>* 1.2**

- `List<E> subList(int firstIncluded, int firstExcluded)`

  returns a list view of the elements within a range of positions.

***java.util.SortedSet<E>* 1.2**

- `SortedSet<E> subSet(E firstIncluded, E firstExcluded)`
- `SortedSet<E> headSet(E firstExcluded)`
- `SortedSet<E> tailSet(E firstIncluded)`

  return a view of the elements within a range.

***java.util.SortedMap<K, V>* 1.2**

- `SortedMap<K, V> subMap(K firstIncluded, K firstExcluded)`
- `SortedMap<K, V> headMap(K firstExcluded)`
- `SortedMap<K, V> tailMap(K firstIncluded)`

  return a map view of the entries whose keys are within a range.

Bulk Operations

So far, most of our examples used an iterator to traverse a collection, one element at a time. However, you can often avoid iteration by using one of the *bulk operations* in the library.

Suppose you want to find the *intersection* of two sets, the elements that two sets have in common. First, make a new set to hold the result.

```
Set<String> result = new HashSet<String>(a);
```

Here, you use the fact that every collection has a constructor whose parameter is another collection that holds the initialization values.

Now, use the `retainAll` method:

```
result.retainAll(b);
```

It retains all elements that also happen to be in `b`. You have formed the intersection without programming a loop.

You can carry this idea further and apply a bulk operation to a *view*. For example, suppose you have a map that maps employee IDs to employee objects and you have a set of the IDs of all employees that are to be terminated.

```
Map<String, Employee> staffMap = . . .;
Set<String> terminatedIDs = . . .;
```

Simply form the key set and remove all IDs of terminated employees.

```
staffMap.keySet().removeAll(terminatedIDs);
```

Because the key set is a view into the map, the keys and associated employee names are automatically removed from the map.

By using a subrange view, you can restrict bulk operations to sublists and subsets. For example, suppose you want to add the first 10 elements of a list to another container. Form a sublist to pick out the first 10:

```
relocated.addAll(staff.subList(0, 10));
```

The subrange can also be a target of a mutating operation.

```
staff.subList(0, 10).clear();
```

Converting Between Collections and Arrays

Because large portions of the Java platform API were designed before the collections framework was created, you occasionally need to translate between traditional arrays and the more modern collections.

If you have an array, you need to turn it into a collection. The `Arrays.asList` wrapper serves this purpose. For example,

```
String[] values = . . .;
HashSet<String> staff = new HashSet<String>(Arrays.asList(values));
```

Obtaining an array from a collection is a bit trickier. Of course, you can use the `toArray` method:

```
Object[] values = staff.toArray();
```

But the result is an array of *objects*. Even if you know that your collection contained objects of a specific type, you cannot use a cast:

```
String[] values = (String[]) staff.toArray(); // Error!
```

The array returned by the `toArray` method was created as an `Object[]` array, and you cannot change its type. Instead, you use a variant of the `toArray` method. Give it an array of length 0 of the type that you'd like. The returned array is then created *as the same array type*:

```
String[] values = staff.toArray(new String[0]);
```

If you like, you can construct the array to have the correct size:

```
staff.toArray(new String[staff.size()]);
```

In this case, no new array is created.

NOTE: You may wonder why you don't simply pass a `Class` object (such as `String.class`) to the `toArray` method. However, this method does "double duty," both to fill an existing array (provided it is long enough) and to create a new array.

java.util.Collection<E> 1.2

- `<T> T[] toArray(T[] array)`
  checks whether the array parameter is larger than the size of the collection. If so, it adds all elements of the collection into the array, followed by a `null` terminator, and it returns the array. If the length of `array` equals the size of the collection, then the method adds all elements of the collection to the array but does not add a `null`

terminator. If there isn't enough room, then the method creates a new array, of the same type as `array`, and fills it with the elements of the collection.

Extending the Framework

The collections framework contains all data structures that most programmers will ever need. However, if you do need a specialized data structure, you can easily extend the framework. As an example, we implement a circular array queue—see Example 2–6.

The framework contains a class `AbstractQueue` that implements all methods of the `Queue` interface except for `size`, `offer`, `poll`, `peek`, and `iterator`. We make use of that class and only implement the missing methods. We then automatically inherit all the remaining methods from the `AbstractQueue` class.

Most of the methods are straightforward, with the exception of the iterator. We implement the queue iterator as an inner class. This enables the iterator methods to access the fields of the enclosing queue object, that is, the object that constructed the iterator. (As we discussed in Volume 1, Chapter 5, every object of a non-static inner class has a reference to the outer class object that created it.)

Also note the protection against concurrent modification. The queue keeps a count of all modifications. When the iterator is constructed, it makes a copy of that count. Whenever the iterator is used, it checks that the counts still match. If not, it throws a `ConcurrentModificationException`.

Example 2–6: CircularArrayQueueTest.java

```
1. import java.util.*;
2.
3. public class CircularArrayQueueTest
4. {
5.    public static void main(String[] args)
6.    {
7.       Queue<String> q = new CircularArrayQueue<String>(5);
8.       q.add("Amy");
9.       q.add("Bob");
10.      q.add("Carl");
11.      q.add("Deedee");
12.      q.add("Emile");
13.      q.remove();
14.      q.add("Fifi");
15.      q.remove();
16.      for (String s : q) System.out.println(s);
17.   }
18. }
19.
20. /**
21.     A first-in, first-out bounded collection.
22. */
23. class CircularArrayQueue<E> extends AbstractQueue<E>
24. {
25.    /**
26.        Constructs an empty queue.
27.        @param capacity the maximum capacity of the queue
28.    */
29.    public CircularArrayQueue(int capacity)
30.    {
31.       elements = (E[]) new Object[capacity];
```

```
32.        count = 0;
33.        head = 0;
34.        tail = 0;
35.     }
36.
37.     public boolean offer(E newElement)
38.     {
39.        assert newElement != null;
40.        if (count < elements.length)
41.        {
42.           elements[tail] = newElement;
43.           tail = (tail + 1) % elements.length;
44.           count++;
45.           modcount++;
46.           return true;
47.        }
48.        else
49.           return false;
50.     }
51.
52.     public E poll()
53.     {
54.        if (count == 0) return null;
55.        E r = elements[head];
56.        head = (head + 1) % elements.length;
57.        count--;
58.        modcount++;
59.        return r;
60.     }
61.
62.     public E peek()
63.     {
64.        if (count == 0) return null;
65.        return elements[head];
66.     }
67.
68.     public int size()
69.     {
70.        return count;
71.     }
72.
73.     public Iterator<E> iterator()
74.     {
75.        return new QueueIterator();
76.
77.     }
78.
79.     private class QueueIterator implements Iterator<E>
80.     {
81.        public QueueIterator()
82.        {
83.           modcountAtConstruction = modcount;
84.        }
85.
86.        public E next()
87.        {
```

```
88.          if (!hasNext()) throw new NoSuchElementException();
89.          E r = elements[(head + offset) % elements.length];
90.          offset++;
91.          return r;
92.       }
93.
94.       public boolean hasNext()
95.       {
96.          if (modcount != modcountAtConstruction)
97.             throw new ConcurrentModificationException();
98.          return offset < elements.length;
99.       }
100.
101.      public void remove()
102.      {
103.         throw new UnsupportedOperationException();
104.      }
105.
106.      private int offset;
107.      private int modcountAtConstruction;
108.   }
109.
110.   private E[] elements;
111.   private int head;
112.   private int tail;
113.   private int count;
114.   private int modcount;
115. }
```

Algorithms

Generic collection interfaces have a great advantage—you only need to implement your algorithms once. For example, consider a simple algorithm to compute the maximum element in a collection. Traditionally, programmers would implement such an algorithm as a loop. Here is how you find the largest element of an array.

```
if (a.length == 0) throw new NoSuchElementException();
T largest = a[0];
for (int i = 1; i < a.length; i++)
   if (largest.compareTo(a[i]) < 0)
      largest = a[i];
```

Of course, to find the maximum of an array list, you would write the code slightly differently.

```
if (v.size() == 0) throw new NoSuchElementException();
T largest = v.get(0);
for (int i = 1; i < v.size(); i++)
   if (largest.compareTo(v.get(i)) < 0)
      largest = v.get(i);
```

What about a linked list? You don't have efficient random access in a linked list, but you can use an iterator.

```
if (l.isEmpty()) throw new NoSuchElementException();
Iterator<T> iter = l.iterator();
T largest = iter.next();
while (iter.hasNext())
{
```

CJ7V2.book Page 131 Wednesday, October 27, 2004 2:32 PM

```
      T next = iter.next();
      if (largest.compareTo(next) < 0)
         largest = next;
   }
```

These loops are tedious to write, and they are just a bit error prone. Is there an off-by-one error? Do the loops work correctly for empty containers? For containers with only one element? You don't want to test and debug this code every time, but you also don't want to implement a whole slew of methods such as these:

```
static <T extends Comparable> T max(T[] a)
static <T extends Comparable> T max(ArrayList<T> v)
static <T extends Comparable> T max(LinkedList<T> l)
```

That's where the collection interfaces come in. Think of the *minimal* collection interface that you need to efficiently carry out the algorithm. Random access with `get` and `set` comes higher in the food chain than simple iteration. As you have seen in the computation of the maximum element in a linked list, random access is not required for this task. Computing the maximum can be done simply by iteration through the elements. Therefore, you can implement the `max` method to take *any* object that implements the `Collection` interface.

```
public static <T extends Comparable> T max(Collection<T> c)
{
   if (c.isEmpty()) throw new NoSuchElementException();
   Iterator<T> iter = c.iterator();
   T largest = iter.next();
   while (iter.hasNext())
   {
      T next = iter.next();
      if (largest.compareTo(next) < 0)
         largest = next;
   }
   return largest;
}
```

Now you can compute the maximum of a linked list, an array list, or an array, with a single method.

That's a powerful concept. In fact, the standard C++ library has dozens of useful algorithms, each of which operates on a generic collection. The Java library is not quite so rich, but it does contain the basics: sorting, binary search, and some utility algorithms.

Sorting and Shuffling

Computer old-timers will sometimes reminisce about how they had to use punched cards and how they actually had to program by hand algorithms for sorting. Nowadays, of course, sorting algorithms are part of the standard library for most programming languages, and the Java programming language is no exception.

The `sort` method in the `Collections` class sorts a collection that implements the `List` interface.

```
List<String> staff = new LinkedList<String>();
// fill collection . . .;
Collections.sort(staff);
```

This method assumes that the list elements implement the `Comparable` interface. If you want to sort the list in some other way, you can pass a `Comparator` object as a second parameter. (We discussed comparators on page 105.) Here is how you can sort a list of items.

```
Comparator<Item> itemComparator = new
   Comparator<Item>()
   {
      public int compare(Item a, Item b)
      {
         return a.partNumber - b.partNumber;
      }
   });
Collections.sort(items, itemComparator);
```

If you want to sort a list in *descending* order, then use the static convenience method `Collections.reverseOrder()`. It returns a comparator that returns `b.compareTo(a)`. For example,

```
Collections.sort(staff, Collections.reverseOrder())
```

sorts the elements in the list `staff` in reverse order, according to the ordering given by the `compareTo` method of the element type. Similarly,

```
Collections.sort(items, Collections.reverseOrder(itemComparator))
```

reverses the ordering of the `itemComparator`.

You may wonder how the `sort` method sorts a list. Typically, when you look at a sorting algorithm in a book on algorithms, it is presented for arrays and uses random element access. However, random access in a list can be inefficient. You can actually sort lists efficiently by using a form of merge sort (see, for example, *Algorithms in C++* by Robert Sedgewick [Addison-Wesley 1998, pp. 366–369]). However, the implementation in the Java programming language does not do that. It simply dumps all elements into an array, sorts the array by using a different variant of merge sort, and then copies the sorted sequence back into the list.

The merge sort algorithm used in the collections library is a bit slower than *quick sort*, the traditional choice for a general-purpose sorting algorithm. However, it has one major advantage: It is *stable*, that is, it doesn't switch equal elements. Why do you care about the order of equal elements? Here is a common scenario. Suppose you have an employee list that you already sorted by name. Now you sort by salary. What happens to employees with equal salary? With a stable sort, the ordering by name is preserved. In other words, the outcome is a list that is sorted first by salary, then by name.

Because collections need not implement all of their "optional" methods, all methods that receive collection parameters must describe when it is safe to pass a collection to an algorithm. For example, you clearly cannot pass an `unmodifiableList` list to the `sort` algorithm. What kind of list *can* you pass? According to the documentation, the list must be modifiable but need not be resizable.

The terms are defined as follows:

- A list is *modifiable* if it supports the `set` method.
- A list is *resizable* if it supports the `add` and `remove` operations.

The `Collections` class has an algorithm `shuffle` that does the opposite of sorting—it randomly permutes the order of the elements in a list. You supply the list to be shuffled and a random number generator. For example,

```
ArrayList<Card> cards = . . .;
Collections.shuffle(cards);
```

If you supply a list that does not implement the `RandomAccess` interface, then the `shuffle` method copies the elements into an array, shuffles the array, and copies the shuffled elements back into the list.

The program in Example 2–7 fills an array list with 49 `Integer` objects containing the numbers 1 through 49. It then randomly shuffles the list and selects the first 6 values from the shuffled list. Finally, it sorts the selected values and prints them.

Example 2–7: ShuffleTest.java

```
1. import java.util.*;
2.
3. /**
4.    This program demonstrates the random shuffle and sort algorithms.
5. */
6. public class ShuffleTest
7. {
8.    public static void main(String[] args)
9.    {
10.      List<Integer> numbers = new ArrayList<Integer>();
11.      for (int i = 1; i <= 49; i++)
12.         numbers.add(i);
13.      Collections.shuffle(numbers);
14.      List<Integer> winningCombination = numbers.subList(0, 6);
15.      Collections.sort(winningCombination);
16.      System.out.println(winningCombination);
17.   }
18. }
```

java.util.Collections 1.2

- `static <T extends Comparable<? super T>> void sort(List<T> elements)`
- `static <T> void sort(List<T> elements, Comparator<? super T> c)`
  sort the elements in the list, using a stable sort algorithm. The algorithm is guaranteed to run in $O(n \log n)$ time, where n is the length of the list.
- `static void shuffle(List<?> elements)`
- `static void shuffle(List<?> elements, Random r)`
  randomly shuffle the elements in the list. This algorithm runs in $O(n\, a(n))$ time, where n is the length of the list and $a(n)$ is the average time to access an element.
- `static <T> Comparator<T> reverseOrder()`
  returns a comparator that sorts elements in the reverse order of the one given by the `compareTo` method of the `Comparable` interface.
- `static <T> Comparator<T> reverseOrder(Comparator<T> comp)`
  returns a comparator that sorts elements in the reverse order of the one given by `comp`.

Binary Search

To find an object in an array, you normally visit all elements until you find a match. However, if the array is sorted, then you can look at the middle element and check whether it is larger than the element that you are trying to find. If so, you keep looking in the first half of the array; otherwise, you look in the second half. That cuts the problem in half. You keep going in the same way. For example, if the array has 1024 elements, you will locate the match (or confirm that there is none) after 10 steps, whereas a linear search would have taken you an average of 512 steps if the element is present, and 1024 steps to confirm that it is not.

The `binarySearch` of the `Collections` class implements this algorithm. Note that the collection must already be sorted or the algorithm will return the wrong answer. To find an element, supply the collection (which must implement the `List` interface—more on that in the note

below) and the element to be located. If the collection is not sorted by the `compareTo` element of the `Comparable` interface, then you must supply a comparator object as well.

```
i = Collections.binarySearch(c, element);
i = Collections.binarySearch(c, element, comparator);
```

A return value of ≥ 0 from the `binarySearch` method denotes the index of the matching object. That is, `c.get(i)` is equal to `element` under the comparison order. If the value is negative, then there is no matching element. However, you can use the return value to compute the location where you *should* insert `element` into the collection to keep it sorted. The insertion location is

```
insertionPoint = -i – 1;
```

It isn't simply `-i` because then the value of 0 would be ambiguous. In other words, the operation

```
if (i < 0)
   c.add(-i - 1, element);
```

adds the element in the correct place.

To be worthwhile, binary search requires random access. If you have to iterate one by one through half of a linked list to find the middle element, you have lost all advantage of the binary search. Therefore, the `binarySearch` algorithm reverts to a linear search if you give it a linked list.

NOTE: JDK 1.3 had no separate interface for an ordered collection with efficient random access, and the `binarySearch` method employed a very crude device, checking whether the list parameter extended the `AbstractSequentialList` class. This has been fixed in JDK 1.4. Now the `binarySearch` method checks whether the list parameter implements the `RandomAccess` interface. If it does, then the method carries out a binary search. Otherwise, it uses a linear search.

java.util.Collections 1.2

- `static <T extends Comparable<? super T>> int binarySearch(List<T> elements, T key)`
- `static <T> int binarySearch(List<T> elements, T key, Comparator<? super T> c)`

  search for a key in a sorted list, using a linear search if `elements` extends the `AbstractSequentialList` class, and a binary search in all other cases. The methods are guaranteed to run in $O(a(n) \log n)$ time, where n is the length of the list and $a(n)$ is the average time to access an element. The methods return either the index of the key in the list, or a negative value `i` if the key is not present in the list. In that case, the key should be inserted at index `-i – 1` for the list to stay sorted.

Simple Algorithms

The `Collections` class contains several simple but useful algorithms. Among them is the example from the beginning of this section, finding the maximum value of a collection. Others include copying elements from one list to another, filling a container with a constant value, and reversing a list. Why supply such simple algorithms in the standard library? Surely most programmers could easily implement them with simple loops. We like the algorithms because they make life easier for the programmer *reading* the code. When you read a loop that was implemented by someone else, you have to decipher the original programmer's intentions. When you see a call to a method such as `Collections.max`, you know right away what the code does.

The following API notes describe the simple algorithms in the `Collections` class.

java.util.Collections 1.2

- static <T extends Comparable<? super T>> T min(Collection<T> elements)
- static <T extends Comparable<? super T>> T max(Collection<T> elements)
- static <T> min(Collection<T> elements, Comparator<? super T> c)
- static <T> max(Collection<T> elements, Comparator<? super T> c)
  return the smallest or largest element in the collection. (The parameter bounds are simplified for clarity.)
- static <T> void copy(List<? super T> to, List<T> from)
  copies all elements from a source list to the same positions in the target list. The target list must be at least as long as the source list.
- static <T> void fill(List<? super T> l, T value)
  sets all positions of a list to the same value.
- static <T> boolean addAll(Collection<? super T> c, T... values) **5.0**
  adds all values to the given collection and returns true if the collection changed as a result.
- static <T> boolean replaceAll(List<T> l, T oldValue, T newValue) **1.4**
  replaces all elements equal to oldValue with newValue.
- static int indexOfSubList(List<?> l, List<?> s) **1.4**
- static int lastIndexOfSubList(List<?> l, List<?> s) **1.4**
  return the index of the first or last sublist of l equalling s, or –1 if no sublist of l equals s. For example, if l is [s, t, a, r] and s is [t, a, r], then both methods return the index 1.
- static void swap(List<?> l, int i, int j) **1.4**
  swaps the elements at the given offsets.
- static void reverse(List<?> l)
  reverses the order of the elements in a list. For example, reversing the list [t, a, r] yields the list [r, a, t]. This method runs in O(n) time, where n is the length of the list.
- static void rotate(List<?> l, int d) **1.4**
  rotates the elements in the list, moving the entry with index i to position (i + d) % l.size(). For example, rotating the list [t, a, r] by 2 yields the list [a, r, t]. This method runs in O(n) time, where n is the length of the list.
- static int frequency(Collection<?> c, Object o) **5.0**
  returns the count of elements in c that equal the object o.
- boolean disjoint(Collection<?> c1, Collection<?> c2) **5.0**
  returns true if the collections have no elements in common.

Writing Your Own Algorithms

If you write your own algorithm (or in fact, any method that has a collection as a parameter), you should work with *interfaces,* not concrete implementations, whenever possible. For example, suppose you want to fill a JMenu with a set of menu items. Traditionally, such a method might have been implemented like this:

```
void fillMenu(JMenu menu, ArrayList<JMenuItem> items)
{
   for (JMenuItem item : items)
      menu.addItem(item);
}
```

However, you now constrained the caller of your method—the caller must supply the choices in an ArrayList. If the choices happen to be in another container, they first need to be repackaged. It is much better to accept a more general collection.

You should ask yourself this: What is the most general collection interface that can do the job? In this case, you just need to visit all elements, a capability of the basic `Collection` interface. Here is how you can rewrite the `fillMenu` method to accept collections of any kind.

```
void fillMenu(JMenu menu, Collection<JMenuItem> items)
{
   for (JMenuItem item : items)
      menu.addItem(item);
}
```

Now, anyone can call this method, with an `ArrayList` or a `LinkedList`, or even with an array, wrapped with the `Arrays.asList` wrapper.

NOTE: If it is such a good idea to use collection interfaces as method parameters, why doesn't the Java library follow this rule more often? For example, the `JComboBox` class has two constructors:

```
JComboBox(Object[] items)
JComboBox(Vector<?> items)
```

The reason is simply timing. The Swing library was created before the collections library.

If you write a method that *returns* a collection, you may also want to return an interface instead of a class because you can then change your mind and reimplement the method later with a different collection.

For example, let's write a method `getAllItems` that returns all items of a menu.

```
List<MenuItem> getAllItems(JMenu menu)
{
   ArrayList<MenuItem> items = new ArrayList<MenuItem>()
   for (int i = 0; i < menu.getItemCount(); i++)
      items.add(menu.getItem(i));
   return items;
}
```

Later, you can decide that you don't want to *copy* the items but simply provide a view into them. You achieve this by returning an anonymous subclass of `AbstractList`.

```
List<MenuItem> getAllItems(final JMenu menu)
{
   return new
      AbstractList<MenuItem>()
      {
         public MenuItem get(int i)
         {
            return item.getItem(i);
         }
         public int size()
         {
            return item.getItemCount();
         }
      };
}
```

Of course, this is an advanced technique. If you employ it, be careful to document exactly which "optional" operations are supported. In this case, you must advise the caller that the returned object is an unmodifiable list.

Legacy Collections

In this section, we discuss the collection classes that existed in the Java programming language since the beginning: the `Hashtable` class and its useful `Properties` subclass, the `Stack` subclass of `Vector`, and the `BitSet` class.

The `Hashtable` *Class*

The classic `Hashtable` class serves the same purpose as the `HashMap` and has essentially the same interface. Just like methods of the `Vector` class, the `Hashtable` methods are synchronized. If you do not require synchronization or compatibility with legacy code, you should use the `HashMap` instead.

NOTE: The name of the class is `Hashtable`, with a lowercase `t`. Under Windows, you'll get strange error messages if you use `HashTable`, because the Windows file system is not case-sensitive but the Java compiler is.

Enumerations

The legacy collections use the `Enumeration` interface for traversing sequences of elements. The `Enumeration` interface has two methods, `hasMoreElements` and `nextElement`. These are entirely analogous to the `hasNext` and `next` methods of the `Iterator` interface.

For example, the `elements` method of the `Hashtable` class yields an object for enumerating the values in the table:

```
Enumeration<Employee> e = staff.elements();
while (e.hasMoreElements())
{
   Employee e = e.nextElement();
   . . .
}
```

You will occasionally encounter a legacy method that expects an enumeration parameter. The static method `Collections.enumeration` yields an enumeration object that enumerates the elements in the collection. For example,

```
ArrayList<InputStream> streams = . . .;
SequenceInputStream in = new SequenceInputStream(Collections.enumeration(streams));
   // the SequenceInputStream constructor expects an enumeration
```

NOTE: In C++, it is quite common to use iterators as parameters. Fortunately, in programming for the Java platform, very few programmers use this idiom. It is much smarter to pass around the collection than to pass an iterator. The collection object is more useful. The recipients can always obtain the iterator from the collection when they need to do so, plus they have all the collection methods at their disposal. However, you will find enumerations in some legacy code because they were the only available mechanism for generic collections until the collections framework appeared in JDK 1.2.

java.util.Enumeration<E> **1.0**

- `boolean hasMoreElements()`
  returns `true` if there are more elements yet to be inspected.
- `E nextElement()`
  returns the next element to be inspected. Do not call this method if `hasMoreElements()` returned `false`.

CJ7V2.book Page 138 Wednesday, October 27, 2004 2:32 PM

java.util.Hashtable<K, V> 1.0

- `Enumeration<K> keys()`
  returns an enumeration object that traverses the keys of the hash table.
- `Enumeration<V> elements()`
  returns an enumeration object that traverses the elements of the hash table.

java.util.Vector<E> 1.0

- `Enumeration<E> elements()`
  returns an enumeration object that traverses the elements of the vector.

Property Sets

A *property set* is a map structure of a very special type. It has three particular characteristics.

- The keys and values are strings.
- The table can be saved to a file and loaded from a file.
- A secondary table for defaults is used.

The Java platform class that implements a property set is called `Properties`.

Property sets are commonly used in specifying configuration options for programs—see Volume 1, Chapter 10.

java.util.Properties 1.0

- `Properties()`
  creates an empty property list.
- `Properties(Properties defaults)`
  creates an empty property list with a set of defaults.
- `String getProperty(String key)`
  gets a property association; returns the string associated with the key, or the string associated with the key in the default table if it wasn't present in the table.
- `String getProperty(String key, String defaultValue)`
  gets a property with a default value if the key is not found; returns the string associated with the key, or the default string if it wasn't present in the table.
- `void load(InputStream in)`
  loads a property set from an `InputStream`.
- `void store(OutputStream out, String commentString)`
  stores a property set to an `OutputStream`.

Stacks

Since version 1.0, the standard library had a `Stack` class with the familiar `push` and `pop` methods. However, the `Stack` class extends the `Vector` class, which is not satisfactory from a theoretical perspective—you can apply such un-stack-like operations as `insert` and `remove` to insert and remove values anywhere, not just at the top of the stack.

java.util.Stack<E> 1.0

- `E push(E item)`
  pushes `item` onto the stack and returns `item`.
- `E pop()`
  pops and returns the top item of the stack. Don't call this method if the stack is empty.

- `E peek()`
  returns the top of the stack without popping it. Don't call this method if the stack is empty.

Bit Sets

The Java platform `BitSet` class stores a sequence of bits. (It is not a *set* in the mathematical sense—bit *vector* or bit *array* would have been more appropriate terms.) Use a bit set if you need to store a sequence of bits (for example, flags) efficiently. Because a bit set packs the bits into bytes, it is far more efficient to use a bit set than to use an `ArrayList` of `Boolean` objects.

The `BitSet` class gives you a convenient interface for reading, setting, or resetting individual bits. Use of this interface avoids the masking and other bit-fiddling operations that would be necessary if you stored bits in `int` or `long` variables.

For example, for a `BitSet` named `bucketOfBits`,

```
bucketOfBits.get(i)
```

returns `true` if the `i`'th bit is on, and `false` otherwise. Similarly,

```
bucketOfBits.set(i)
```

turns the `i`'th bit on. Finally,

```
bucketOfBits.clear(i)
```

turns the `i`'th bit off.

C++ NOTE: The C++ `bitset` template has the same functionality as the Java platform `BitSet`.

`java.util.BitSet` 1.0

- `BitSet(int initialCapacity)`
  constructs a bit set.
- `int length()`
  returns the "logical length" of the bit set: 1 plus the index of the highest set bit.
- `boolean get(int bit)`
  gets a bit.
- `void set(int bit)`
  sets a bit.
- `void clear(int bit)`
  clears a bit.
- `void and(BitSet set)`
  logically ANDs this bit set with another.
- `void or(BitSet set)`
  logically ORs this bit set with another.
- `void xor(BitSet set)`
  logically XORs this bit set with another.
- `void andNot(BitSet set)`
  clears all bits in this bit set that are set in the other bit set.

The "Sieve of Eratosthenes" Benchmark

As an example of using bit sets, we want to show you an implementation of the "sieve of Eratosthenes" algorithm for finding prime numbers. (A prime number is a number like 2,

3, or 5 that is divisible only by itself and 1, and the sieve of Eratosthenes was one of the first methods discovered to enumerate these fundamental building blocks.) This isn't a terribly good algorithm for finding the number of primes, but for some reason it has become a popular benchmark for compiler performance. (It isn't a good benchmark either, because it mainly tests bit operations.)

Oh well, we bow to tradition and include an implementation. This program counts all prime numbers between 2 and 2,000,000. (There are 148,933 primes, so you probably don't want to print them all out.)

Without going into too many details of this program, the key is to march through a bit set with 2 million bits. We first turn on all the bits. After that, we turn off the bits that are multiples of numbers known to be prime. The positions of the bits that remain after this process are themselves the prime numbers. Example 2–8 illustrates this program in the Java programming language, and Example 2–9 is the C++ code.

NOTE: Even though the sieve isn't a good benchmark, we couldn't resist timing the two implementations of the algorithm. Here are the timing results on a 1.7-GHz IBM ThinkPad with 1 GB of RAM, running Red Hat Linux 9.

C++ (g++ 3.2.2): 330 milliseconds

Java (JDK 5.0): 105 milliseconds

We have run this test for six editions of *Core Java*, and in the last three editions Java easily beat C++. In previous editions, we pointed out, in all fairness, that the culprit for the bad C++ result is the lousy performance of the standard `bitset` template. When we reimplemented `bitset`, the time for C++ used to be faster than Java. Not anymore—with a handcrafted bitset, the C++ time was 140 milliseconds in our latest experiment.

Example 2–8: Sieve.java

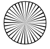
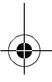
```
1. import java.util.*;
2.
3. /**
4.    This program runs the Sieve of Eratosthenes benchmark.
5.    It computes all primes up to 2,000,000.
6. */
7. public class Sieve
8. {
9.    public static void main(String[] s)
10.    {
11.       int n = 2000000;
12.       long start = System.currentTimeMillis();
13.       BitSet b = new BitSet(n + 1);
14.       int count = 0;
15.       int i;
16.       for (i = 2; i <= n; i++)
17.          b.set(i);
18.       i = 2;
19.       while (i * i <= n)
20.       {
21.          if (b.get(i))
22.          {
23.             count++;
24.             int k = 2 * i;
25.             while (k <= n)
```

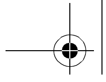
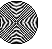
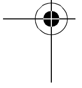
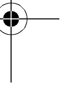
CJ7V2.book Page 141 Wednesday, October 27, 2004 2:32 PM

```
26.             {
27.                b.clear(k);
28.                k += i;
29.             }
30.          }
31.          i++;
32.       }
33.       while (i <= n)
34.       {
35.          if (b.get(i))
36.             count++;
37.          i++;
38.       }
39.       long end =  System.currentTimeMillis();
40.       System.out.println(count + " primes");
41.       System.out.println((end - start) + " milliseconds");
42.    }
43. }
```

Example 2–9: Sieve.cpp

```
1. #include <bitset>
2. #include <iostream>
3. #include <ctime>
4.
5. using namespace std;
6.
7. int main()
8. {
9.    const int N = 2000000;
10.   clock_t cstart = clock();
11.
12.   bitset<N + 1> b;
13.   int count = 0;
14.   int i;
15.   for (i = 2; i <= N; i++)
16.      b.set(i);
17.   i = 2;
18.   while (i * i <= N)
19.   {
20.      if (b.test(i))
21.      {
22.         count++;
23.         int k = 2 * i;
24.         while (k <= N)
25.         {
26.            b.reset(k);
27.            k += i;
28.         }
29.      }
30.      i++;
31.   }
32.   while (i <= N)
33.   {
34.      if (b.test(i))
35.         count++;
36.      i++;
```

```
37.     }
38.
39.     clock_t cend = clock();
40.     double millis = 1000.0
41.        * (cend - cstart) / CLOCKS_PER_SEC;
42.
43.     cout << count << " primes\n"
44.        << millis << " milliseconds\n";
45.
46.     return 0;
47. }
```