Chapter 2

Collections

- Collection Interfaces
- Concrete Collections
- The Collections Framework
- Algorithms
- Legacy Collections

OOP encapsulates data inside classes, but this doesn't make how 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 arraylike 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, there is a course called *Data Structures* that 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 the Java 2 platform, 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 the Java 2 platform, 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.* The Java library does not supply a queue, but it is nevertheless a good example to introduce the basic concepts.

NOTE: If you need a queue, you can simply use the `LinkedList` class that we discuss later in this chapter.

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).

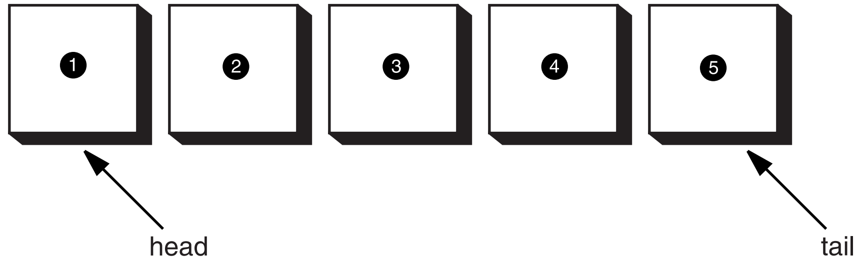
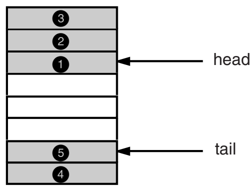


Figure 2–1: A queue

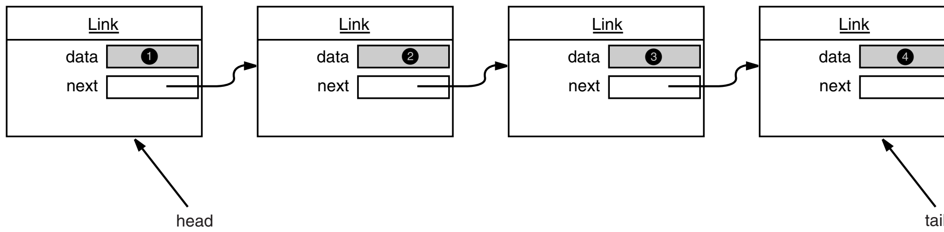
If there was a queue interface in the collections library, it might look like this:

```
interface Queue
{  void add(Object obj);
   Object remove();
   int size();
}
```

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



Circular Array



Linked List

Figure 2–2: Queue implementations

Each implementation can be expressed by a class that realizes the `Queue` interface:

```
class CircularArrayQueue implements Queue
{  CircularArrayQueue(int capacity) { . . . }
   public void add(Object obj) { . . . }
   public Object remove() { . . . }
   public int size() { . . . }

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

class LinkedListQueue implements Queue
{  LinkedListQueue() { . . . }
   public void add(Object obj) { . . . }
   public Object remove() { . . . }
   public int size() { . . . }

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

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 (such as `CircularArrayQueue`) *only* when you construct the collection object. Use the *interface type* to hold the collection reference.

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

This approach makes it easy to change your mind and 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 expressLane = new LinkedListQueue();
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.

This example illustrates another issue for the designer of a collection class library. Strictly speaking, in a bounded collection, the interface of the add method should indicate that the method can fail:

```
class CircularArrayQueue
{  public void add(Object obj)
      throws CollectionFullException
   . . .
}
```

That's a problem—now the CircularArrayQueue class can't implement the Queue interface since you can't add exception specifiers when overriding a method. Should one have two interfaces, BoundedQueue and Queue? Or should the add method throw an unchecked exception? There are advantages and disadvantages to both approaches. It is these kinds of issues that make it genuinely hard to design a logically coherent collection class library.

As we already mentioned, the Java library has no separate class for queues. We just used this example to illustrate the difference between interface and implementation since a queue has a simple interface and two well-known distinct implementations. In the next section, you will see how the Java library classifies the collections that it supports.

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:

```
boolean add(Object obj)
Iterator iterator()
```

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

The add method adds an object to the collection. The add method returns true if adding the object actually changed the collection; 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 is rejected since sets reject duplicates.

The iterator method returns an object that implements the Iterator interface—we will describe that interface in a moment. You can use the iterator object to visit the elements in the container one by one.

The Iterator interface has three fundamental methods:

```
Object 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 container, you request an iterator and then keep calling the `next` method while `hasNext` returns true.

```
Iterator iter = c.iterator();
while (iter.hasNext())
{  Object obj = iter.next();
   do something with obj
}
```

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 extend the `Enumeration` interface. But they disliked the cumbersome method names and chose to introduce a new interface with shorter method names instead.

Finally, the `remove` method removes the element that was returned by the last call to `next`.

You may well wonder why the `remove` method is a part of the `Iterator` interface. It is more efficient to remove an element if you know *where* it is. The iterator knows about *positions* in the collection. Therefore, the `remove` method was added to the `Iterator` interface. If you visited an element that you didn't like, you can efficiently remove it.

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, just like you can advance an array index with the `i++` operation 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).

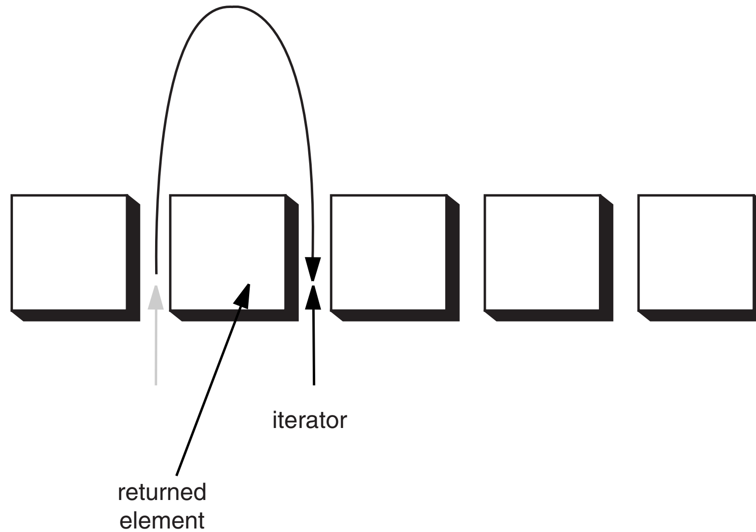


Figure 2–3: Advancing an iterator

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.

You must be careful when using the `remove` method. Calling `remove` removes the element that was returned by the last call to `next`. That makes sense if you want to remove a particular value—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 by position, you first need to skip past the element. For example, here is how you remove the first element in a collection.

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

More importantly, 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
```

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 `print` method that prints all elements in a collection.

```
public static void print(Collection c)
{  System.out.print("[ ");
   Iterator iter = c.iterator();
   while (iter.hasNext())
      System.out.print(iter.next() + " ");
   System.out.println("]");
}
```

NOTE: We give this example to illustrate how to write a generic method. If you want to print the elements in a collection, you can just call `System.out.println(c)`. This works because each collection class has a `toString` method that returns a string containing all elements in the collection.

Here is a method that adds all objects from one collection to another:

```
public static boolean addAll(Collection to, Collection from)
{  Iterator iter = from.iterator();
   boolean modified = false;
   while (iter.hasNext())
      if (to.add(iter.next()))
         modified = true;
   return modified;
}
```

Recall that the `add` method returns `true` if adding the element modified the collection. You can implement these utility methods for arbitrary collections because the `Collection` and `Iterator` interfaces supply fundamental methods such as `add` and `next`.

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, users don't have to keep reinventing the wheel. The `addAll` method is one such method.

Had `Collection` been an abstract class instead of an interface, then it would have been an easy matter to supply this functionality in the class. However, you cannot supply methods in interfaces of the Java programming language. Therefore, the collection library takes a slightly different approach. 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 from)
boolean remove(Object obj)
boolean removeAll(Collection c)
void clear()
boolean retainAll(Collection c)
Object[] toArray()
```

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 class `AbstractCollection` leaves the fundamental methods (such as `add` and `iterator`) abstract but implements the routine methods in terms of them. For example,

```
public class AbstractCollection
   implements Collection
{  . . .
   public abstract boolean add(Object obj);

   public boolean addAll(Collection from)
   {  Iterator iter = iterator();
      boolean modified = false;
      while (iter.hasNext())
         if (add(iter.next()))
            modified = true;
      return modified
   }
   . . .
}
```

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

- `Iterator 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`.

  Parameters: `obj` the object to match in the collection
- `boolean containsAll(Collection other)`
  returns `true` if this collection contains all elements in the other collection.

  Parameters: `other` the collection holding the elements to match
- `boolean add(Object element)`
  adds an element to the collection. Returns `true` if the collection changed as a result of this call.

  Parameters: `element` the element to add
- `boolean addAll(Collection other)`
  adds all elements from the other collection to this collection. Returns `true` if the collection changed as a result of this call.

  Parameters: `other` the collection holding the elements to add
- `boolean remove(Object obj)`
  removes an object equal to `obj` from this collection. Returns `true` if a matching object was removed.

  Parameters: `obj` an object that equals the element to remove
- `boolean removeAll(Collection other)`
  removes all elements from the other collection from this collection. Returns `true` if the collection changed as a result of this call.

  Parameters: `other` the collection holding the elements to add
- `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.

Parameters: `other` the collection holding the elements to be kept

- `Object[] toArray()`
  returns an array of the objects in the collection.

`java.util.Iterator`

- `boolean hasNext()`
  returns `true` if there is another element to visit.
- `Object next()`
  returns the next object to visit. Throws a `NoSuchElementException` if the end of the collection has been reached.
- `Object remove()`
  removes and returns 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 you have a thorough understanding of what classes you will want to use, we will return to abstract considerations and see how the collections framework organizes these classes.

Linked Lists

We used arrays and their dynamic cousin, the `Vector` class, for many examples in Volume 1. However, arrays and vectors suffer from a major drawback. Removing an element from the middle of an array is very 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.

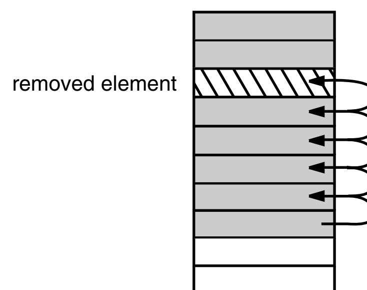


Figure 2–4: Removing an element from an array

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).

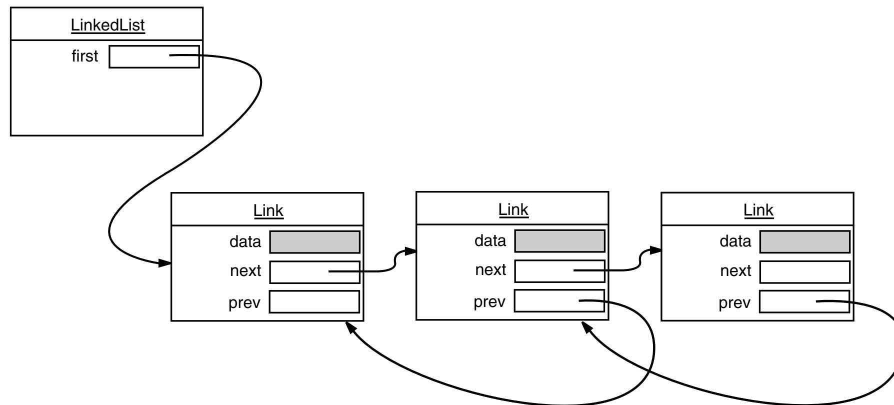


Figure 2–5: A doubly linked list

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).

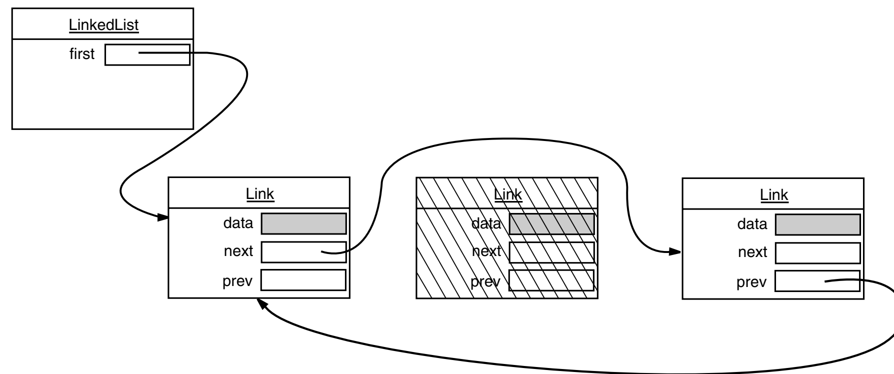


Figure 2–6: Removing an element from a linked list

Perhaps you once had a course in data structures where 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 `LinkedList` class implements the `Collection` interface. You can use the familiar methods to traverse a list. The following code example prints the first three elements of a list, adds three elements, and then removes the third one.

```
LinkedList staff = new LinkedList();
staff.add("Angela");
staff.add("Bob");
staff.add("Carl");
Iterator iter = staff.iterator();
for (int i = 0; i < 3; i++)
   System.out.println(iter.next());
iter.remove(); // remove last visited element
```

However, there is an important difference between linked lists and generic collections. A linked list is an *ordered collection* where 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 only makes sense 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 extends Iterator
{  void add(Object);
   . . .
}
```

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

In addition, the `ListIterator` interface has two methods—

```
Object previous()
boolean hasPrevious()
```

—that you can use for traversing a list backwards. 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 iter = staff.listIterator();
```

The `add` method adds the new element *before* the iterator position. For example, the code

```
ListIterator iter = staff.listIterator();
iter.next();
iter.add("Juliet");
```

skips past the first element in the linked list and adds `"Juliet"` before the second element (see Figure 2–7).

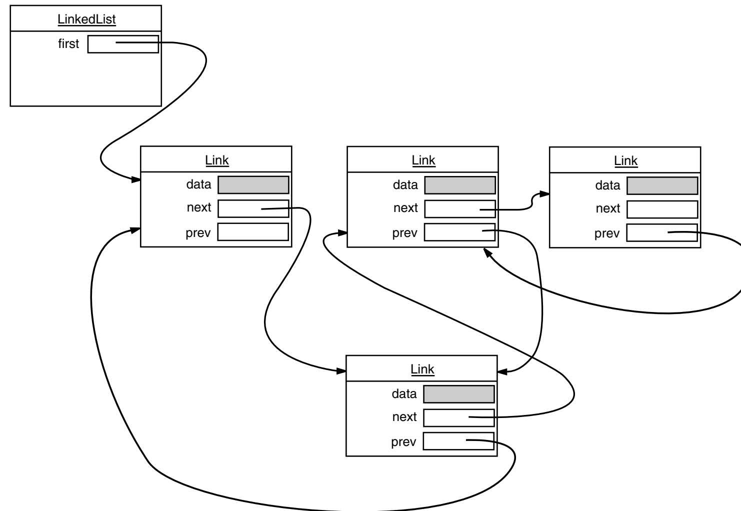


Figure 2–7: Adding an element to a linked list

If you call the `add` method multiple times, the elements are simply added in the order in which you supplied them. They all get 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 the four possible positions (marked as `|`) for inserting a new element are:

```
|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 only depends on the iterator position, the `remove` method depends on the iterator state.

Finally, there is a `set` method that 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 iter = list.listIterator();
Object 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:

```
LinkedList list = . . .;
ListIterator iter1 = list.listIterator();
ListIterator 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 container 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 container 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 container. 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, however, that it does not automatically make collections safe for multithreading. We discuss thread safety issues later in this chapter.

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, there are many other useful methods for operating on linked lists that 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"`. However, there is no method that returns an iterator to that position. If you want to do something with the element beyond knowing that it exists, you have to program an iteration loop by hand.

CAUTION: The Java platform documentation points out that you should not add a reference of a collection to itself. Otherwise, it is easy to generate a stack overflow in the virtual machine. For example, the following call is fatal:

```
LinkedList list = new LinkedList();
list.add(list); // add list to itself
String contents = list.toString(); // dies with infinite recursion
```

Naturally, this is not a situation that comes up in everyday programming.

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 where elements need to be accessed by an integer index.

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

```
Object 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)`. Of course, 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 only have a few elements, you can just use an array or a collection such as `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`.

Example 2–1: LinkedListTest.java

```
import java.util.*;

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

```
   {  List a = new LinkedList();
      a.add("Angela");
      a.add("Carl");
      a.add("Erica");

      List b = new LinkedList();
      b.add("Bob");
      b.add("Doug");
      b.add("Frances");
      b.add("Gloria");

      // merge the words from b into a

      ListIterator aIter = a.listIterator();
      Iterator bIter = b.iterator();

      while (bIter.hasNext())
      {  if (aIter.hasNext()) aIter.next();
         aIter.add(bIter.next());
      }

      System.out.println(a);

      // remove every second word from b

      bIter = b.iterator();
      while (bIter.hasNext())
      {  bIter.next(); // skip one element
         if (bIter.hasNext())
         {  bIter.next(); // skip next element
            bIter.remove(); // remove that element
         }
      }

      System.out.println(b);

      // bulk operation: remove all words in b from a

      a.removeAll(b);

      System.out.println(a);

   }
}
```

`java.util.List`

- `ListIterator listIterator()`
  returns a list iterator for visiting the elements of the list.
- `ListIterator 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.

Parameters:	`index`	the position of the next visited element

- `void add(int i, Object element)`
  adds an element at the specified position.

Parameters:	`index`	the position of the new element
	`element`	the element to add

- `void addAll(int i, Collection elements)`
  adds all elements from a collection to the specified position.

Parameters:	`index`	the position of the first new element
	`elements`	the elements to add

- `Object remove(int i)`
  removes and returns an element at the specified position.

Parameters:	`index`	the position of the element to remove

- `Object set(int i, Object element)`
  replaces the element at the specified position with a new element and returns the old element.

Parameters:	`index`	the replacement position
	`element`	the new 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.

Parameters:	`element`	the element to match

- `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.

Parameters:	`element`	the element to match

java.util.ListIterator

- `void add(Object element)`
  adds an element before the current position.
  Parameters: `element` the element to add
- `void set(Object element)`
  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`.
  Parameters: `element` the new element
- `boolean hasPrevious()`
  returns `true` if there is another element to visit when iterating backwards through the list.
- `Object 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`

- `LinkedList()`
  constructs an empty linked list.
- `LinkedList(Collection elements)`
  constructs a linked list and adds all elements from a collection.
  Parameters: `elements` the elements to add
- `void addFirst(Object element)`
- `void addLast(Object element)`
  add an element to the beginning or the end of the list.
  Parameters: `element` the element to add
- `Object getFirst()`
- `Object getLast()`
  return the element at the beginning or the end of the list.

- `Object removeFirst()`
- `Object 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 are not appropriate for linked lists, but of course they make a lot of sense for arrays. The collections library supplies an `ArrayList` class that implements the `List` interface. An `ArrayList` is similar to a `Vector`: it encapsulates a dynamically reallocated `Object[]` array.

Why use an `ArrayList` instead of a `Vector`? There is 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 only access a vector from 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.

Using an `ArrayList` is as simple as using a `Vector`. Just keep in mind that you need to use the short method names `get` and `set` instead of the `elementAt` and `setElementAt` methods.

Hash Sets

Linked lists and arrays let you specify in which order 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. We see in the next section how these hash codes are computed. What's important for now is that hash codes can be computed quickly and that the computation only depends 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 345 and there are 101 buckets, then the object is placed in bucket 42 (because the remainder of the integer division 345/101 is 42). 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 need to 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.

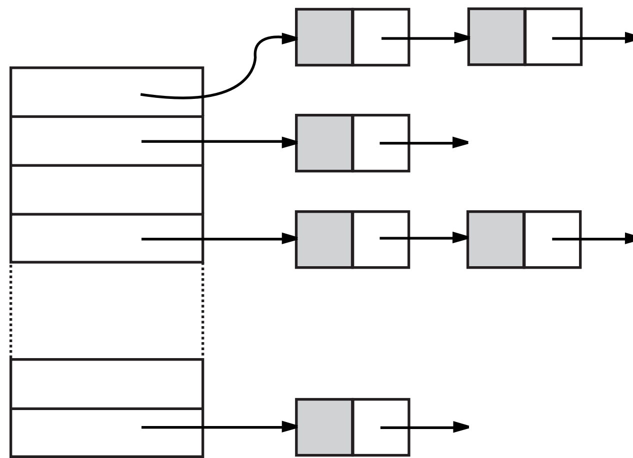


Figure 2–8: A hash table

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 should set the initial bucket count to about 150 percent of the expected element count. Some researchers believe that it is a good idea to make the size of the hash table a prime number to prevent a clustering of keys. The evidence for this isn't conclusive, but it certainly can't hurt. For example, if you need to store about 100 entries, set the initial bucket size to 151.

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. In the Java programming language, 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 hash table becomes more than 75 percent full, then the table is automatically rehashed, using 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 only adds it if it is not yet present.

The Java collections library supplies a `HashSet` class that implements a set based on a hash table. At the time of this writing, the default constructor `HashSet` constructs a hash table with 101 buckets and a load factor of 0.75. These values may change in future releases. If you at all care about these values, you should specify your own, with the constructors

```
HashSet(int initialCapacity)
HashSet(int initialCapacity, float loadFactor)
```

You add elements with the `add` method. The `contains` method is redefined to make a quick lookup to find if an element is already present in the set. It only checks the elements in one bucket and not all elements in the collection. The hash set iterator visits all buckets in turn. Since the hashing scatters the elements around in the table, they are visited in seemingly random order. You would only use a hash set 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 `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.

Example 2–2: SetTest.java

```
import java.util.*;
import java.io.*;

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

```
   {  Set words = new HashSet(59999);
         // set to HashSet or TreeSet
      long totalTime = 0;

      try
      {  BufferedReader in = new
            BufferedReader(new InputStreamReader(System.in));
         String line;
         while ((line = in.readLine()) != null)
         {  StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens())
            {  String word = tokenizer.nextToken();
               long callTime = System.currentTimeMillis();
               words.add(word);
               callTime = System.currentTimeMillis() - callTime;
               totalTime += callTime;
            }
         }
      }
      catch (IOException e)
      {  System.out.println("Error " + e);
      }

      Iterator iter = words.iterator();
      while (iter.hasNext())
         System.out.println(iter.next());
      System.out.println(words.size()
         + " distinct words. " + totalTime + " milliseconds.");
   }
}
```

`java.util.HashSet`

- `HashSet()`
  constructs an empty hash set.
- `HashSet(Collection elements)`
  constructs a hash set and adds all elements from a collection.

  Parameters: `elements` the elements to add
- `HashSet(int initialCapacity)`
  constructs an empty hash set with the specified capacity.

  Parameters: `initialCapacity` the initial number of buckets
- `HashSet(int initialCapacity, float loadFactor)`
  constructs an empty hash set with the specified capacity and load factor.

Parameters:	`initialCapacity`	the initial number of buckets
	`loadFactor`	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

Hash functions

You can insert strings into a hash set because the `String` class has a `hashCode` method that computes a *hash code* for a string. A hash code is an integer that is somehow derived from the characters in the string. Table 2–1 lists a few examples of hash codes that result from the `hashCode` function of the `String` class.

Table 2–1: Hash codes resulting from the `hashCode` function

String	Hash code
Hello	140207504
Harry	140013338
Hacker	884756206

The `hashCode` method is defined in the `Object` class. Therefore, every object has a default hash code. That hash code is derived from the object's memory address. In general, the default hash function is not very useful because objects with identical contents may yield different hash codes. Consider this example.

```
String s = "Ok";
StringBuffer sb = new StringBuffer(s);
System.out.println(s.hashCode() + " " + sb.hashCode());
String t = "Ok";
StringBuffer tb = new StringBuffer(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

Table 2–2 shows the result.

Table 2–2: Hash codes of objects with identical contents

`"Ok" String` hash code	`"Ok" StringBuffer` hash code
`3030`	`20526976`
`3030`	`20527144`

Note that the strings `s` and `t` have the same hash value because, for strings, the hash values are derived from their *contents.* The string buffers `sb` and `tb` have different hash values because no special hash function has been defined for the `StringBuffer` class and the default hash code function in the `Object` class derives the hash code from the object's memory address.

You should always define the hashCode method for objects that you insert into a hash table. This method should return an integer (which can be negative). The hash table code will later reduce the integer by dividing by the bucket count and taking the remainder. Just scramble up the hash codes of the data fields in some way that will give the hash codes for different objects a good chance of being widely scattered.

For example, suppose you have the class Item for inventory items. An item consists of a description string and a part number.

```
anItem = new Item("Toaster", 49954);
```

If you want to construct a hash set of items, you need to define a hash code. For example,

```
class Item
{   . . .
   public int hashCode()
   {   return 13 * description.hashCode() + 17 * partNumber;
   }
   . . .
   private String description;
   private int partNumber;
}
```

As a practical matter, if the part number uniquely identifies the item, you don't need to incorporate the hash code of the description.

Furthermore, you *also* need to make sure that the equals method is well defined. The Object class defines equals, but that method only tests whether or not two objects are *identical*. If you don't redefine equals, then every new object that you insert into the table will be considered a *different* object.

You need to redefine equals to check for equal contents.

```
class Item
{  . . .
     public boolean equals(Object other)
     {  if (other != null && getClass() == other.getClass())
        {  Item otherItem = (Item)other;
           return description.equals(otherItem.description)
               && partNumber == otherItem.partNumber;
        }
        else
           return false;
     }
     . . .
}
```

CAUTION: Your 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()`.

`java.lang.Object`

- `boolean equals(Object obj)`
  compares two objects for equality; returns `true` if both objects are equal; `false` otherwise.

Parameters:	`obj`	the object to compare with the first object (may be `null`, in which case the method should return `false`)

- `int hashCode()`
  returns a hash code for this object. A hash code can be any integer, positive or negative. Equal objects need to return identical hash codes.

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.

```
TreeSet sorter = new TreeSet();
sorter.add("Bob");
sorter.add("Angela");
sorter.add("Carl");
Iterator iter = sorter.iterator();
while (iter.hasNext()) System.println(iter.next());
```

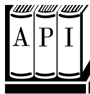
Then, the values are printed in sorted order: `Angela Bob Carl`. As the name of the class suggests, the sorting is accomplished by a tree data structure. (The current implementation uses a *red-black tree.* For a detailed description of red-black trees, see, for example, *Introduction to Algorithms* by Thomas Cormen, Charles Leiserson, and Ronald Rivest [The MIT Press 1990].) 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`

- `TreeSet()`
  constructs an empty tree set.
- `TreeSet(Collection elements)`
  constructs a tree set and adds all elements from a collection.
  Parameters: `elements` the elements to add

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:

```
int compareTo(Object 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 need to 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
{  public int compareTo(Object other)
   {   Item otherItem = (Item)other;
       return partNumber - otherItem.partNumber;
   }
   . . .
}
```

Note that the explicit argument of the compareTo method has type Object, not Comparable. If the object is not of the correct type, then this compareTo method simply throws a ClassCastException. (The compareTo methods in the standard library behave in the same way when presented with illegal argument types.)

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 does *not* work if the integers can be negative. 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. You can only implement the interface 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 has a single method, with two explicit parameters:

```
int compare(Object a, Object 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
{  public int compare(Object a, Object b)
   {  Item itemA = (Item)a;
      Item itemB = (Item)b;
      String descrA = itemA.getDescription();
      String descrB = itemB.getDescription();
      return descrA.compareTo(descrB);
   }
}
```

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

```
ItemComparator comp = new ItemComparator();
TreeSet sortByDescription = new TreeSet(comp);
```

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

Note that the 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:

```
TreeSet sortByDescription = new TreeSet(
   new Comparator()
   {  public int compare(Object a, Object b)
      {  Item itemA = (Item)a;
         Item itemB = (Item)b;
         String descrA = itemA.getDescription();
         String descrB = itemB.getDescription();
         return descrA.compareTo(descrB);
      }
   });
```

Using comparators, you can sort elements in any way you wish.

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 importantly, 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, using a custom comparator.

Example 2–3: TreeSetTest.java

```
import java.util.*;

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

```
   {  SortedSet parts = new TreeSet();
      parts.add(new Item("Toaster", 1234));
      parts.add(new Item("Widget", 4562));
      parts.add(new Item("Modem", 9912));
      System.out.println(parts);

      SortedSet sortByDescription = new TreeSet(
         new Comparator()
         {  public int compare(Object a, Object b)
            {  Item itemA = (Item)a;
               Item itemB = (Item)b;
               String descrA = itemA.getDescription();
               String descrB = itemB.getDescription();
               return descrA.compareTo(descrB);
            }
         });

      sortByDescription.addAll(parts);
      System.out.println(sortByDescription);
   }
}

class Item implements Comparable
{  public Item(String aDescription, int aPartNumber)
   {  description = aDescription;
      partNumber = aPartNumber;
   }

   public String getDescription()
   {  return description;
   }

   public String toString()
   {  return "[description=" + description
         + ", partNumber=" + partNumber + "]";
   }

   public boolean equals(Object other)
   {  if (getClass() == other.getClass())
      {  Item otherItem = (Item)other;
         return description.equals(otherItem.description)
            && partNumber == otherItem.partNumber;
      }
      else
         return false;
   }

   public int hashCode()
```

```
    {   return 13 * description.hashCode() + 17 * partNumber;
    }

    public int compareTo(Object other)
    {   Item otherItem = (Item)other;
        return partNumber - otherItem.partNumber;
    }

    private String description;
    private int partNumber;
}
```

java.lang.Comparable

- `int compareTo(Object 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`.

  Parameters: `other` the object to compare

java.util.Comparator

- `int compare(Object a, Object 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`.

  Parameters: `a, b` the objects to compare

java.util.SortedSet

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

java.util.TreeSet

- `TreeSet(Comparator c)`
  constructs a tree set and uses the specified comparator for sorting its elements.

  Parameters: `c` the comparator to use for sorting

- `TreeSet(SortedSet elements)`
  constructs a tree set, adds all elements from a sorted set, and uses the same element comparator as the given sorted set.

Parameters:	`elements`	the sorted set with the elements to add and the comparator to use

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`. 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.

```
HashMap staff = new HashMap();
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 = "1411-16-2536";
e = (Employee)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. (This feature is useful; if `put` returns a non-`null` value, then you know you replaced a previous entry.)

The `remove()` method removes an element 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 keySet()
Collection values()
Set entrySet()
```

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

Note that the `keySet` is *not* a `HashSet` or `TreeSet`, but it is an object of some other class that implements the `Set` interface. We discuss the `Set` interface and its purpose in detail in the next section. The `Set` interface extends the `Collection` interface. In fact, as you will see, it does not add any new methods. Therefore, you can use it exactly as you use the `Collection` interface.

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

```
Set keys = map.keySet();
Iterator iter = keys.iterator();
while (iter.hasNext())
{  Object key = iter.next();
   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:

```
Set entries = staff.entrySet();
Iterator iter = entries.iterator();
while (iter.hasNext())
{  Map.Entry entry = (Map.Entry)iter.next();
   Object key = entry.getKey();
   Object 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

key/value set view has the same restriction, even though it would make conceptual sense to add a new key/value pair.

NOTE: The legacy `Hashtable` class (which we cover later in this chapter) has methods that return enumeration objects—the classical analog to iterators—that traverse keys and values. However, having collection views is more powerful since they let you operate on all keys or values at once.

Example 2–4 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 2–4: MapTest.java

```
import java.util.*;

public class MapTest
{  public static void main(String[] args)
   {  Map staff = new HashMap();
      staff.put("144-25-5464", new Employee("Angela Hung"));
      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"));

      // print all entries

      System.out.println(staff);

      // remove an entry

      staff.remove("567-24-2546");

      // replace an entry

      staff.put("456-62-5527", new Employee("Francesca Miller"));

      // look up a value

      System.out.println(staff.get("157-62-7935"));

      // iterate through all entries

      Set entries = staff.entrySet();
      Iterator iter = entries.iterator();
      while (iter.hasNext())
      {  Map.Entry entry = (Map.Entry)iter.next();
```

```
         Object key = entry.getKey();
         Object value = entry.getValue();
         System.out.println("key=" + key + ", value=" + value);
      }
   }
}

class Employee
{  public Employee(String n)
   {  name = n;
      salary = 0;
   }

   public String toString()
   {  return "[name=" + name + ", salary=" + salary + "]";
   }

   public void setSalary(double s)
   {  salary = s;
   }

   private String name;
   private double salary;
}
```

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 since 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 onto a queue. The operations of the `WeakHashMap` periodically check that queue for newly arrived weak references. When a weak reference arrives in the queue, this is an indication that the key was no longer used by anyone and that it has been collected. The `WeakHashMap` then removes the associated entry.

java.util.Map

- `Object get(Object 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.
  Parameters: `key` the key to use for retrieval (may be `null`)
- `Object put(Object key, Object 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.
  Parameters: `key` the key to use for retrieval (may be `null`)
  `value` the associated object (may not be `null`)
- `void putAll(Map entries)`
  adds all entries from the specified map to this map.
  Parameters: `entries` the map with the entries to be added
- `boolean containsKey(Object key)`
  returns `true` if the key is present in the map.
  Parameters: `key` the key to find
- `boolean containsValue(Object value)`
  returns `true` if the value is present in the map.
  Parameters: `value` the value to find
- `Set 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 keySet()`
  returns a set view of all keys in the map. You can remove elements from this set, and the key and associated values are removed from the map, but you cannot add any elements.
- `Collection 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.

API

`java.util.Map.Entry`

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

Parameters:	`value`	the new value to associate with the key

API

`java.util.HashMap`

- `HashMap()`
  constructs an empty hash map.
- `HashMap(Map entries)`
  constructs a hash map and adds all entries from a map.

Parameters:	`entries`	the entries to add

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

Parameters:	`initialCapacity`	the initial number of buckets
	`loadFactor`	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 is 0.75

`java.util.WeakHashMap`

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

Parameters:	`initialCapacity`	the initial number of buckets
	`loadFactor`	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 is 0.75

`java.util.SortedMap`

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

`java.util.TreeMap`

- `TreeMap(Comparator c)`
  constructs a tree set and uses the specified comparator for sorting its keys.

Parameters:	`c`	the comparator to use for sorting

- `TreeMap(Map entries)`
  constructs a tree map and adds all entries from a map.

Parameters:	`entries`	the entries to add

- `TreeMap(SortedMap entries)`
  constructs a tree set, adds all entries from a sorted map, and uses the same element comparator as the given sorted map.

Parameters:	`entries`	the sorted set with the entries to add and the comparator to use

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 sublasses 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 2–9), 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, then it is helpful to understand the framework.

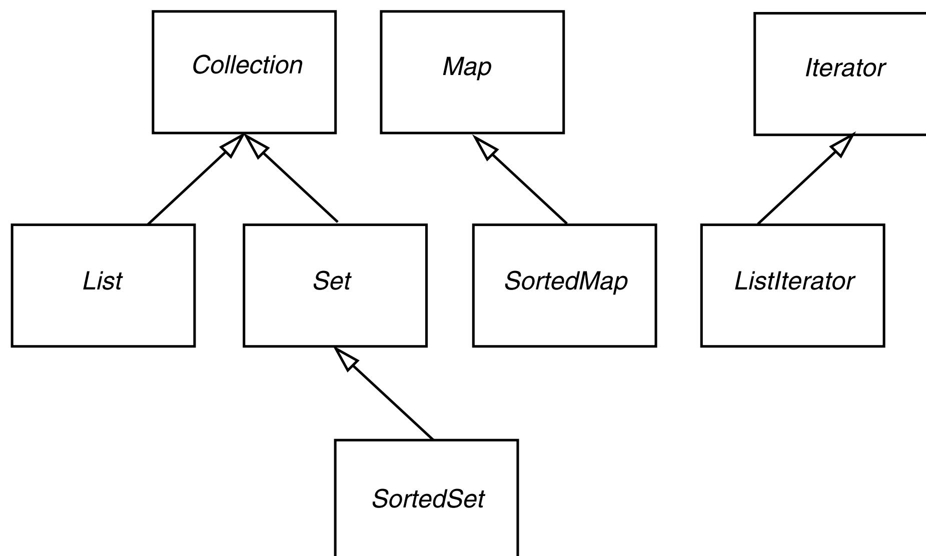


Figure 2–9: The interfaces of the collections framework

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

```
boolean add(Object element)
```

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

```
boolean put(Object key, Object 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:

```
Object get(Object 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, Object element)
Object get(int index)
void remove(int index)
```

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

```
void add(Object element)
```

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

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 was 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. 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 `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.

NOTE: For sets and lists, there is a well-defined notion of equality. Two sets are equal if they contain the same elements in some order. Two lists are equal if they contain the same elements in the same order. However, there is no well-defined notion of equality for collections. You should therefore not use the `equals` method on `Collection` references.

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 only accept 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 containers. 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. There are five abstract classes that supply many of these routine implementations:

```
AbstractCollection
AbstractList
AbstractSequentialList
AbstractSet
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 six concrete classes:

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

Figure 2–10 shows the relationships between these classes.

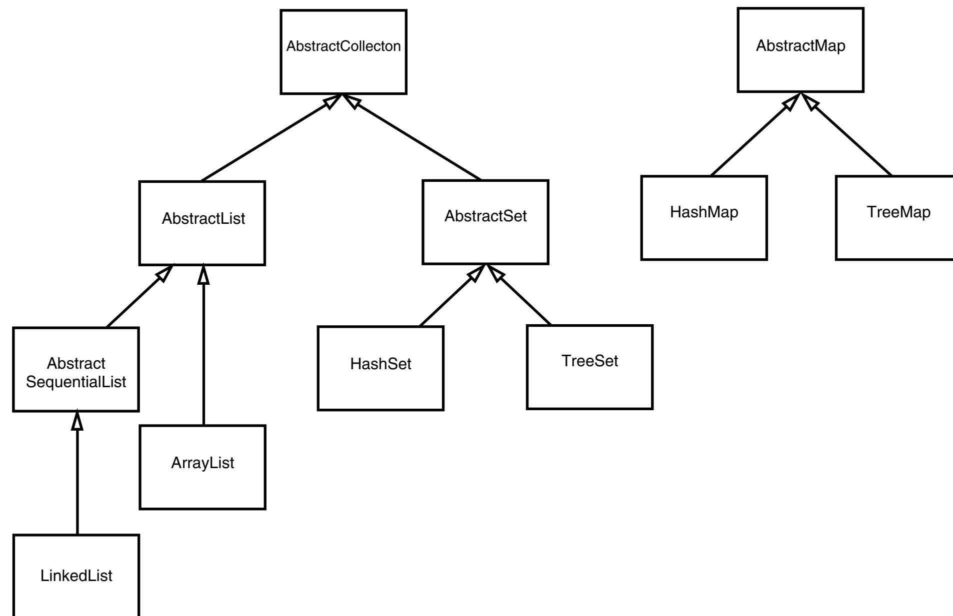


Figure 2–10: Classes in the collections framework

Finally, there are a number of "legacy" container classes that have been present since the beginning, before there was a collections framework:

```
Vector
Stack
Hashtable
Properties
```

They have been integrated into the collections framework—see Figure 2–11. We discuss these classes later in this chapter.

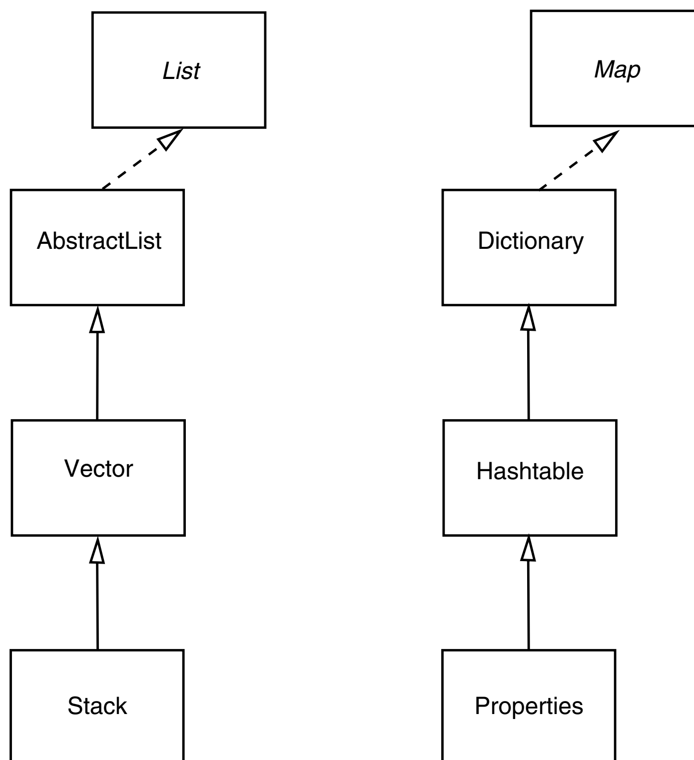


Figure 2–11: Legacy classes in the collections framework

Views and Wrappers

If you look at Figure 2–9 and Figure 2–10, you might think it is overkill to have six interfaces and five abstract classes to implement six 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.* You have no way of knowing, and you need not know, exactly what class the library uses to implement the view.

The technique of views has a number of useful applications in the collections framework. Here is the most compelling one. Recall that the methods of the `Vector` class are synchronized, which is unnecessarily slow if the vector is only accessed from a single thread. For that reason, we recommended the use of the `ArrayList` instead of the `Vector` class. However, if you do access a collection from multiple threads, it is very important that the methods are synchronized. For example, it would be disastrous if one thread tried to add to a hash table while another thread is rehashing the elements. The library designers could have implemented companion classes with synchronized methods for all data structures. But they did something more useful. They supplied a mechanism that produces synchronized views for all *interfaces.* For example, the static `synchronizedMap` class in the `Collections` class can turn any `Map` into a `Map` with synchronized access methods:

```
HashMap hashMap = new HashMap();
map = Collections.synchronizedMap(hashMap);
```

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

There are six methods to obtain synchronized collections:

```
Collections.synchronizedCollection
Collections.synchronizedList
Collections.synchronizedSet
Collections.synchronizedSortedSet
Collections.synchronizedMap
Collections.synchronizedSortedMap
```

The views that are returned by these methods are sometimes called *wrappers.*

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, but to simply pass the constructed collection to the wrapper method:

```
map = Collections.synchronizedMap(new HashMap());
```

There is one caveat when accessing a collection from multiple threads. Recall that you can either have multiple iterators that read from a collection or have a single thread that modifies the collection. For that reason, you will want to make sure that any iteration—

```
Iterator iter = collection.iterator();
while (iter.hasNext())
   do something with iter.next();
```

—does not occur at a time when another thread modifies the collection. If it did, then the `next` method would throw a `ConcurrentModificationException`. One way to ensure exclusive access is to place the iteration inside a block that locks the container:

```
synchronized (container)
{  Iterator iter = collection.iterator();
   while (iter.hasNext())
      do something with iter.next();
}
```

Since the wrappers 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 synchronization wrapper.

CAUTION: The `synchronizedCollection` method (as well as the `unmodifiableCollection` method 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 wrapper acts in this way because equality testing is not well defined at this level of the hierarchy.

However, the `synchronizedSet` and `synchronizedList` class do not hide the `equals` methods of the underlying collections.

The wrappers treat the `hashCode` method in the same way.

The `Collections` class has another potentially useful set of wrappers to produce unmodifiable views of collections. These wrappers 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.

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 staff = new LinkedList();
. . .
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. There are similar methods to obtain unmodifiable wrappers to the other collection interfaces.

NOTE: In the API documentation, certain methods of the collection interfaces, such as add, are described as "optional operations." This is curious—isn't the purpose of an interface to lay out the methods that a class *must* implement? Indeed, it would have made sense to separate the read-only interfaces from the interfaces that allow full access. But that would have doubled the number of interfaces, which the designers of the library found unacceptable.

The Arrays class has a static asList method that 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 cardList = Arrays.asList(cardDeck);
```

The returned object is *not* an ArrayList. It is a view object whose get and set methods 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.

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.

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

```
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, there are similar methods

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

that return views into the maps consisting of all entries where the *keys* fall into the specified ranges.

Lightweight collection wrappers

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`. There is very little storage cost—the object is only stored once. This is a cute application of the wrapper technique. You can use such a list to initialize a concrete container. For example, the following call creates an `ArrayList` containing 100 strings, all set to `"DEFAULT"`:

```
ArrayList settings
   = new ArrayList(Collections.nCopies(100, "DEFAULT"));
```

The method call

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

returns a wrapper 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 a hash table or tree. The constants `Collections.EMPTY_LIST` and `Collections.EMPTY_SET` return objects that implement the `List` and `Set` interfaces and contain no elements. The advantage is similar to that of the `singleton` method: the returned objects do not have the overhead of a data structure. Singletons and empty objects are potentially useful as parameters to methods that expect a list, set, or container.

NOTE: JDK 1.3 adds methods `singletonList` and `singletonMap` and a constant `EMPTY_MAP`.

A final note on optional operations

We'd like to end this section with a few words about the "optional" or unsupported operations in the collection and iterator interfaces. This is undoubtedly the most controversial design decision in the collections framework. The problem is caused by the undeniably powerful and convenient views. Views are good because they lead to efficient code, and their "plug and play" nature means you only need

to learn a few basic concepts. 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. The restriction is different for each view. Making a separate interface for each restricted view would lead to a bewildering tangle of interfaces that would be unusable in practice.

Should you extend the technique of "optional" methods to your own interfaces? We think not. Even though collections are used very 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. Look at a few other libraries, such as the JGL library from ObjectSpace (`www.objectspace.com`), to see a different set of trade-offs. Or, even better, try your hand at designing your own library of collections and algorithms. You will soon run into the inevitable conflicts and feel much more sympathy with the folks from Sun. 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`

- `static Collection synchronizedCollection(Collection c)`
- `static List synchronizedList(List c)`
- `static Set synchronizedSet(Set c)`
- `static SortedSet synchronizedSortedSet(SortedSet c)`
- `static Map synchronizedMap(Map c)`
- `static SortedMap synchronizedSortedMap(SortedMap c)`

  construct a view of the collection whose methods are synchronized.

  Parameters: `c` the collection to wrap

- `static Collection unmodifiableCollection(Collection c)`
- `static List unmodifiableList(List c)`
- `static Set unmodifiableSet(Set c)`
- `static SortedSet unmodifiableSortedSet(SortedSet c)`
- `static Map unmodifiableMap(Map c)`
- `static SortedMap unmodifiableSortedMap(SortedMap c)`

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

  Parameters: `c` the collection to wrap

- `static List nCopies(int n, Object value)`
- `static Set singleton(Object value)`

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

Parameters:	`n`	the number of times to repeat the value in the list
	`value`	the element value in the collection

- `static final List EMPTY_LIST`
- `static final Set EMPTY_SET`

An unmodifiable wrapper for an empty list or set.

`java.util.Arrays`

- `static List asList(Object[] array)`

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

Parameters:	`array`	the array to wrap

`java.util.List`

- `List subList(int from, int to)`

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

Parameters:	`from`	the first position to include in the view
	`to`	the first position to exclude in the view

`java.util.SortedSet`

- `SortedSet subSet(Object from, Object to)`
- `SortedSet headSet(Object to)`
- `SortedSet tailSet(Object from)`

return a view of the elements within a range.

Parameters:	`from`	the first element to include in the view
	`to`	the first element to exclude in the view

`java.util.SortedMap`

- `SortedMap subMap(Object from, Object to)`
- `SortedMap headMap(Object to)`
- `SortedMap tailMap(Object from)`

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

Parameters:	`from`	the first key to include in the view
	`to`	the first key to exclude in the view

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 i*n 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 result = new HashSet(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 staffMap = . . .;
Set 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 ten elements of a list to another container. Form a sublist to pick out the first ten:

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

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

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

Interfacing with Legacy APIs

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

First, consider the case where you have values in an array or vector and you want to put them into a collection. If the values are inside a `Vector`, simply construct your collection from the vector:

```
Vector values = . . .;
HashSet staff = new HashSet(values);
```

All collection classes have a constructor that can take an arbitrary collection object. Since the Java 2 platform, the `Vector` class implements the `List` interface.

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

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

Conversely, if you need to call a method that requires a vector, you can construct a vector from any collection:

```
Vector values = new Vector(staff);
```

Obtaining an array 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 need to use a variant of the `toArray` method. Give it an array of length 0 of the type that you'd like. Then, the returned array is created as the same array type, and you can cast it:

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

NOTE: You may wonder why you don't simply pass a `Class` object (such as `String.class`) to the `toArray` method. However, as you can see from the API notes, 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

- `Object[] toArray(Object[] array)`

  checks if 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 the incoming array*, and fills it with the elements of the collection.

Parameters:	`array`	the array that holds the collection elements, or whose element type is used to create a new array to hold the collection elements

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();
Comparable 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 a vector, the code would be slightly different.

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

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

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

```
Object max(Comparable[] a)
Object max(Vector v)
Object max(LinkedList 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 iterating through the elements. Therefore, you can implement the `max` method to take *any* object that implements the `Collection` interface.

```
public static Object max(Collection c)
{  if (c.isEmpty()) throw new NoSuchElementException();
   Iterator iter = c.iterator();
```

```
    Comparable largest = (Comparable)iter.next();
    while (iter.hasNext())
    {  Comparable next = (Comparable)iter.next();
       if (largest.compareTo(next) < 0) largest = next;
    }
    return largest;
 }
```

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

```
LinkedList l;
Vector v;
Employee[] a;
. . .
largest = max(l);
largest = max(v);
largest = max(Arrays.asList(a));
```

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 sorting algorithms by hand. 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 staff = new LinkedList();
// 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 in the section on sorted sets.) Here is how you can sort a list of employees by increasing salary.

```
Collections.sort(staff,
   new Comparator()
   {  public compare(Object a, Object b)
      {  double salaryDifference = (Employee)a.getSalary()
            - (Employee)b.getSalary();
         if (salaryDifference < 0) return -1;
         if (salaryDifference > 0) return 1;
         return 0;
```

```
    }
  });
```

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)`. (The objects must implement the `Comparable` interface.) 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.

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. But 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++, Parts 1–4,* by Robert Sedgwick [Addison-Wesley 1998, p. 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, 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 need to 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.

These 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 cards = . . .;
Collections.shuffle(cards);
```

The current implementation of the `shuffle` algorithm requires random access to the list elements, so it won't work too well with a large linked list.

The program in Example 2–5 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 out.

Example 2–5: ShuffleTest.java

```
import java.util.*;

public class ShuffleTest
{  public static void main(String[] args)
   {  List numbers = new ArrayList(49);
      for (int i = 1; i <= 49; i++)
         numbers.add(new Integer(i));
      Collections.shuffle(numbers);
      List winningCombination = numbers.subList(0, 6);
      Collections.sort(winningCombination);
      System.out.println(winningCombination);
   }
}
```

`java.util.Collections`

- `static void sort(List elements)`
- `static void sort(List elements, Comparator 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.

Parameters:	`elements`	the list to sort
	`c`	the comparator to use for sorting

- `static void shuffle(List elements)`
- `static void shuffle(List elements, Random r)`
  randomly shuffles 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.

Parameters:	`elements`	the list to shuffle
	`r`	the source of randomness for shuffling

- `static Comparator reverseOrder()`
  returns a comparator that sorts elements in the reverse order of the one given by the `compareTo` method of the `Comparable` interface.

Binary Search

To find an object in an array, you normally need to visit all elements until you find a match. However, if the array is sorted, then you can look at the middle element and check if 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 need to supply a comparator object as well.

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

If the return value of the `binarySearch` method is ≥ 0, it 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: Unfortunately, since there is no separate interface for an ordered collection with efficient random access, the `binarySearch` method employs a very crude device to find out whether to carry out a binary or a linear search. It checks whether the list parameter implements the `AbstractSequentialList` class. If it does, then the parameter is certainly a linked list, because the abstract sequential list is a skeleton implementation of a linked list. In all other cases, the `binarySearch` algorithm makes the assumption that the collection supports efficient random access and proceeds with a binary search.

java.util.Collections

- `static int binarySearch(List elements, Object key)`
- `static int binarySearch(List elements, Object key, Comparator c)`

  search for a key in a sorted list, using a linear search if `elements` extends the `AbstractSequentialList` class, 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.

Parameters:	`elements`	the list to search
	`key`	the object to find
	`c`	the comparator used for sorting the list elements

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

- `static Object min(Collection elements)`
- `static Object max(Collection elements)`
- `static Object min(Collection elements, Comparator c)`
- `static Object max(Collection elements, Comparator c)`

  return the smallest or largest element in the collection.

Parameters:	`elements`	the collection to search
	`c`	the comparator used for sorting the elements

- `static void copy(List to, List 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.

Parameters:	`to`	the target list
	`from`	the source list

- `static void fill(List l, Object value)`
  sets all positions of a list to the same value.

Parameters:	`l`	the list to fill
	`value`	the value with which to fill the list

- `static void reverse(List l)`
  reverses the order of the elements in a list. This method runs in O(n) time, where n is the length of the list.

Parameters:	`l`	the list to reverse

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 `JComboBox` with a set of strings. Traditionally, such a method might have been implemented like this:

```
void fillComboBox(JComboBox comboBox, Vector choices)
{  for (int i = 0; i < choices.size(); i++)
      comboBox.addItem(choices.get(i));
}
```

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

You should ask yourself 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 `fillComboBox` method to accept collections of any kind.

```
void fillComboBox(JComboBox comboBox, Collection choices)
{  Iterator iter = choices.iterator();
   while (iter.hasNext())
      comboBox.addItem(iter.next());
}
```

Now, anyone can call this method with a vector 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. You should expect future APIs to rely more heavily on the collections library. In particular, vectors should be "on their way out" because of the synchronization overhead.

If you write a method that *returns* a collection, you don't have to change the return type to a collection interface. The user of your method might in fact have a slight preference to receive the most concrete class possible. However, for your own convenience, you may 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 combo box. You could simply return the collection that you used to gather the items, say, an `ArrayList`.

```
ArrayList getAllItems(JComboBox comboBox)
{  ArrayList items = new ArrayList(comboBox.getItemCount());
   for (int i = 0; i < comboBox.getItemCount(); i++)
      items.set(i, comboBox.getItemAt(i));
   return items;
}
```

Or, you could change the return type to `List`.

```
List getAllItems(JComboBox comboBox)
```

Then, you are free to change the implementation later. For example, you may decide that you don't want to *copy* the elements of the combo box but simply provide a view into them. You achieve this by returning an anonymous subclass of `AbstractList`.

```
List getAllItems(final JComboBox comboBox)
{  return new
      AbstractList()
      {  public Object get(int i)
         {  return comboBox.getItemAt(i);
         }
         public int size()
```

```
            {  return comboBox.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 e = staff.elements();
while (e.hasMoreElements())
{   Employee e = (Employee)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,

```
ArraySet streams = . . .; // a sequence of input 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 it when they need it, plus they have all the collection methods at their disposal. However, you will find enumerations in some legacy code since they were the only available mechanism for generic collections until the collections framework appeared in the Java 2 platform.

`java.util.Enumeration`

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

`java.util.Hashtable`

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

`java.util.Vector`

- `Enumeration 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.
- There is a secondary table for defaults.

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

Property sets are useful in specifying configuration options for programs. The environment variables in UNIX and DOS are good examples. On a PC, your `AUTOEXEC.BAT` file might contain the settings:

```
SET PROMPT=$p$g
SET TEMP=C:\Windows\Temp
SET CLASSPATH=c:\jdk\lib;.
```

Here is how you would model those settings as a property set in the Java programming language.

```
Properties settings = new Properties();
settings.put("PROMPT", "$p$g");
settings.put("TEMP", "C:\\Windows\\Temp");
settings.put("CLASSPATH", "c:\\jdk\\lib;.");
```

Use the `store` method to save this list of properties to a file. Here, we just print the property set to the standard output. The second argument is a comment that is included in the file.

```
settings.store(System.out, "Environment settings");
```

The sample table gives the following output.

```
#Environment settings
#Sun Jan 21 07:22:52  1996
CLASSPATH=c:\\jdk\\lib;.
TEMP=C:\\Windows\\Temp
PROMPT=$p$g
```

System information

Here's another example of the ubiquity of the `Properties` set: information about your system is stored in a `Properties` object that is returned by a method of the `System` class. Applications have complete access to this information, but applets that are loaded from a Web page do not—a security exception is thrown if they try to access certain keys. The following code prints out the key/value pairs in the `Properties` object that stores the system properties.

```
import java.util.*;

public class SystemInfo
{  public static void main(String args[])
   {   Properties systemProperties = System.getProperties();
       Enumeration enum = systemProperties.propertyNames();
       while (enum.hasMoreElements())
       {   String key = (String)enum.nextElement();
           System.out.println(key + "=" +
              systemProperties.getProperty(key));
       }
   }
}
```

Here is an example of what you would see when you run the program. You can see all the values stored in this `Properties` object. (What you would get will, of course, reflect your machine's settings):

```
java.specification.name=Java Platform API Specification
awt.toolkit=sun.awt.windows.WToolkit
java.version=1.2.1
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
user.timezone=America/Los_Angeles
java.specification.version=1.2
java.vm.vendor=Sun Microsystems Inc.
user.home=C:\WINDOWS
java.vm.specification.version=1.0
os.arch=x86
java.awt.fonts=
java.vendor.url=http://java.sun.com/
user.region=US
file.encoding.pkg=sun.io
java.home=C:\JDK1.2.1\JRE
java.class.path=.
line.separator=

java.ext.dirs=C:\JDK1.2.1\JRE\lib\ext
java.io.tmpdir=C:\WINDOWS\TEMP\
os.name=Windows 95
java.vendor=Sun Microsystems Inc.
java.awt.printerjob=sun.awt.windows.WPrinterJob
java.vm.specification.vendor=Sun Microsystems Inc.
sun.io.unicode.encoding=UnicodeLittle
file.encoding=Cp1252
java.specification.vendor=Sun Microsystems Inc.
user.language=en
user.name=Cay
java.vendor.url.bug=http://java.sun.com/cgi-bin/bugreport.cgi
java.vm.name=Classic VM
java.class.version=46.0
java.vm.specification.name=Java Virtual Machine Specification
sun.boot.library.path=C:\JDK1.2.1\JRE\bin
os.version=4.10
java.vm.version=1.2.1
java.vm.info=build JDK-1.2.1-A, native threads, symcjit
java.compiler=symcjit
path.separator=;
file.separator=\
user.dir=C:\temp
```

NOTE: For security reasons, applets can only access a small subset of these properties.

Property defaults

A property set is also a useful gadget whenever you want to allow the user to customize an application. Here is how your users can customize the `NotHelloWorld` program to their hearts' content. We'll allow them to specify the following in the configuration file `CustomWorld.ini`:

- window size
- font
- point size
- background color
- message string

If the user doesn't specify some of the settings, we will provide defaults.

The `Properties` class has two mechanisms for providing defaults. First, whenever you look up the value of a string, you can specify a default that should be used automatically when the key is not present.

```
String font = settings.getProperty("FONT", "Courier");
```

If there is a `"FONT"` property in the property table, then `font` is set to that string. Otherwise, `font` is set to `"Courier"`.

If you find it too tedious to specify the default in every call to `getProperty`, then you can pack all the defaults into a secondary property set and supply that in the constructor of your lookup table.

```
Properties defaultSettings = new Properties();
defaultSettings.put("FONT", "Courier");
defaultSettings.put("SIZE", "10");
defaultSettings.put("MESSAGE", "Hello, World");
. . .
Properties settings = new Properties(defaultSettings);
FileInputStream sf = new FileInputStream("CustomWorld.ini");
settings.load(sf);
. . .
```

Yes, you can even specify defaults to defaults if you give another property set parameter to the `defaultSettings` constructor, but it is not something one would normally do.

Figure 2–12 is the customizable `"Hello World"` program. Just edit the `.ini` file to change the program's appearance to the way *you* want (see Figure 2–12).

Figure 2–12: The customized Hello World program

Here are the current property settings.

```
#Environment settings
#Sun Jan 21 07:22:52  1996
FONT=Times New Roman
SIZE=400 200
MESSAGE=Hello, Custom World
COLOR=0 50 100
PTSIZE=36
```

NOTE: The `Properties` class *extends* the `Hashtable` class. That means, all methods of `Hashtable` are available to `Properties` objects. Some functions are useful. For example, `size` returns the number of possible properties (well, it isn't *that* nice—it doesn't count the defaults). Similarly, `keys` returns an enumeration of all keys, except for the defaults. There is also a second function, called `propertyNames`, that returns all keys. The `put` function is downright dangerous. It doesn't check that you put strings into the table.

Does the *is-a* rule for using inheritance apply here? Is every property set a hash table? Not really. That these are true is really just an implementation detail. Maybe it is better to think of a property set as having a hash table. But then the hash table should be a private data field. Actually, in this case, a property set uses two hash tables, one for the defaults and one for the nondefault values.

We think a better design would be the following:

```
class Properties
{  public String getProperty(String) { . . . }
   public void put(String, String) { . . . }
   . . .
   private Hashtable nonDefaults;
   private Hashtable defaults;
}
```

We don't want to tell you to avoid the `Properties` class in the Java library. Provided you are careful to put nothing but strings in it, it works just fine. But think twice before using "quick and dirty" inheritance in your own programs.

Example 2–6: CustomWorld.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import javax.swing.*;

public class CustomWorld
{  public static void main(String[] args)
   {  JFrame frame = new CustomWorldFrame();
      frame.show();
   }
}

class CustomWorldFrame extends JFrame
{  public CustomWorldFrame()
   {  addWindowListener(new WindowAdapter()
         {  public void windowClosing(WindowEvent e)
            {  System.exit(0);
            }
         } );

      Properties defaultSettings = new Properties();
      defaultSettings.put("FONT", "Monospaced");
      defaultSettings.put("SIZE", "300 200");
      defaultSettings.put("MESSAGE", "Hello, World");
      defaultSettings.put("COLOR", "0 50 50");
      defaultSettings.put("PTSIZE", "12");

      Properties settings = new Properties(defaultSettings);
      try
      {  FileInputStream sf
            = new FileInputStream("CustomWorld.ini");
         settings.load(sf);
      }
      catch (FileNotFoundException e) {}
      catch (IOException e) {}

      StringTokenizer st = new StringTokenizer
         (settings.getProperty("COLOR"));
      int red = Integer.parseInt(st.nextToken());
      int green = Integer.parseInt(st.nextToken());
      int blue = Integer.parseInt(st.nextToken());
```

```
      Color foreground = new Color(red, green, blue);

      String name = settings.getProperty("FONT");
      int size = Integer.parseInt(settings.getProperty("PTSIZE"));
      Font f = new Font(name, Font.BOLD, size);

      st = new StringTokenizer(settings.getProperty("SIZE"));
      int hsize = Integer.parseInt(st.nextToken());
      int vsize = Integer.parseInt(st.nextToken());
      setSize(hsize, vsize);
      setTitle(settings.getProperty("MESSAGE"));

      getContentPane().add(new HelloWorldPanel(getTitle(),
         foreground, f), "Center");
   }
}

class HelloWorldPanel extends JPanel
{  public HelloWorldPanel(String aMessage, Color aForeground,
      Font aFont)
   {  message = aMessage;
      foreground = aForeground;
      font = aFont;
   }

   public void paintComponent(Graphics g)
   {  super.paintComponent(g);
      g.setColor(foreground);
      g.setFont(font);

      FontMetrics fm = g.getFontMetrics(font);
      int w = fm.stringWidth(message);

      Dimension d = getSize();
      int cx = (d.width - w) / 2;
      int cy = (d.height + fm.getHeight()) / 2 - fm.getDescent();

      g.drawString(message, cx, cy);
   }

   private Color foreground;
   private Font font;
   private String message;
}
```

`java.util.Properties`

- `Properties()`
  creates an empty property list.
- `Properties(Properties defaults)`
  creates an empty property list with a set of defaults.
  Parameters: `defaults` the defaults to use for lookups
- `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.
  Parameters: `key` the key whose associated string to get
- `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.
  Parameters: `key` the key whose associated string to get
  `defaultValue` the string to return if the key is not present
- `void load(InputStream in) throws IOException`
  loads a property set from an `InputStream`.
  Parameters: `in` the input stream

`java.util.Stack`

- `void push(Object item)`
  pushes an item onto the stack.
  Parameters: `item` the item to be added
- `Object pop()`
  pops and returns the top item of the stack. Don't call this method if the stack is empty.
- `Object 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,

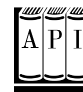
```
bucketOfBits.set(i)
```

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`

- `BitSet(int nbits)`
  constructs a bit set.

  Parameters: `nbits` the initial number of bits

- `int length()`
  returns the "logical length" of the bit set: one plus the index of the highest set bit.

- `boolean get(int bit)`
  gets a bit.

  Parameters: `bit` the position of the requested bit

- `void set(int bit)`
  sets a bit.

  Parameters: `bit` the position of the bit to be set

- `void clear(int bit)`
  clears a bit.

  Parameters: `bit` the position of the bit to be cleared

- `void and(BitSet set)`
  logically ANDs this bit set with another.

Parameters: `set` the bit set to be combined with this bit set

- `void or(BitSet set)`

  logically ORs this bit set with another.

  Parameters: `set` the bit set to be combined with this bit set

- `void xor(BitSet set)`

  logically XORs this bit set with another.

  Parameters: `set` the bit set to be combined with this bit set

- `void andNot(BitSet set)`

  clears all bits in this bitset that are set in the other bit set..

  Parameters: `set` the bit set to be combined with this 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, since it mainly tests bit operations.)

Oh well, we bow to tradition and include an implementation. This program counts all prime numbers between 2 and 1,000,000. (There are 78,498 primes, so you probably don't want to print them all out.) You will find that the program takes a little while to get going, but eventually it picks up speed.

Without going into too many details of this program, the key is to march through a bit set with one 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 2–7 illustrates this program in the Java programming language, and Example 2–8 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 Pentium-166 with 96 megabytes of RAM, running Windows 98.

Borland C++ 5.4: 3750 milliseconds

JDK 1.2.1: 1640 milliseconds

We have run this test for four editions of Core Java, and this is the first time that the Java programming language beat C++. However, in all fairness, we should point out that the culprit for the bad C++ result is the lousy implementation of the standard `bitset` tem-

plate in the Borland compiler. When we reimplemented `bitset`, the time for C++ went down to 1090 milliseconds.

Of course, these are perfect benchmark results because they allow you to put on any spin that you like. If you want to "prove" that the Java programming language is 50 percent slower than C++, make use of the latter results. Or you can "prove" that it has now overtaken C++. Point out that it is only fair to compare the language implementation as a whole, including standard class libraries, and quote the first set of numbers.

Example 2–7: Sieve.java

```
import java.util.*;

public class Sieve
{  public static final boolean PRINT = false;

   public static void main(String[] s)
   {  int n = 1000000;
      long start = System.currentTimeMillis();
      BitSet b = new BitSet(n);
      int count = 0;
      int i;
      for (i = 2; i <= n; i++)
         b.set(i);
      i = 2;
      while (i * i <= n)
      {  if (b.get(i))
         {  if (PRINT) System.out.println(i);
            count++;
            int k = 2 * i;
            while (k <= n)
            {  b.clear(k);
               k += i;
            }
         }
         i++;
      }
      while (i <= n)
      {  if (b.get(i))
         {  if (PRINT) System.out.println(i);
            count++;
         }
         i++;
      }
      long end =  System.currentTimeMillis();
      System.out.println(count + " primes");
      System.out.println((end - start) + " milliseconds");
   }
}
```

Example 2–8: Sieve.cpp

```
#ifndef AVOID_STANDARD_BITSET

#include <bitset>

#else

template<int N>
class bitset
{
public:
   bitset() : bits(new char[(N - 1) / 8 + 1]) {}

   bool test(int n)
   {  return (bits[n >> 3] & (1 << (n & 7))) != 0;
   }

   void set(int n)
   {  bits[n >> 3] |= 1 << (n & 7);
   }

   void reset(int n)
   {  bits[n >> 3] &= ~(1 << (n & 7));
   }

private:
   char* bits;
};

#endif

#include <iostream>
#include <ctime>

using namespace std;

int main()
{  const int N = 1000000;
   clock_t cstart = clock();

   bitset<N + 1> b;
   int count = 0;
   int i;
   for (i = 2; i <= N; i++)
      b.set(i);
   i = 2;
   while (i * i <= N)
   {  if (b.test(i))
```

```
      {  int k = 2 * i;
         while (k <= N)
         {  b.reset(k);
            k += i;
         }
      }
      i++;
   }
   for (i = 2; i <= N; i++)
   {  if (b.test(i))
      {
#ifdef PRINT
         cout << i << "\n";
#endif
         count++;
      }
   }

   clock_t cend = clock();
   double millis = 1000.0
      * (cend - cstart) / CLOCKS_PER_SEC;

   cout << count << " primes\n"
      << millis << " milliseconds\n";

   return 0;
}
```