CHAPTER 9

Lists

While the Set interface is the simplest of the collection types (as it doesn't add any behavior to the Collection interface), you're more likely to use the List interface and its implementations: ArrayList and LinkedList. Beyond the basic collection capabilities, the List interface adds positional access to a collection so its elements have some kind of sequential order. Thus, when retrieving elements out of a list, instead of using an Iterator to visit each element, you can fetch a ListIterator to take advantage of that positioning in order to move in both directions.

The class hierarchy diagram for the List interface, its abstract and concrete implementations, and its related interfaces are shown in Figure 9-1.

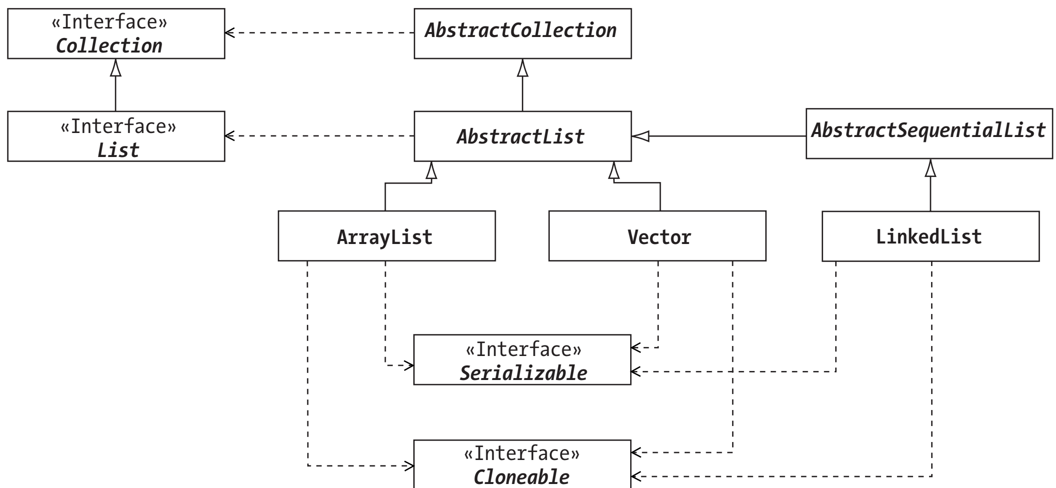

Figure 9-1. The List class hierarchy.

List Basics

Originally demonstrated with the historical Vector class in Chapter 3, lists in Java permit ordered access of their elements. They can have duplicates and because their lookup key is the position and not some hash code, every element can be modified while they remain in the list.

Table 9-1 lists the methods of the List interface. Added to those in the Collection interface are methods supporting element access by position or order.

Table 9-1. Summary of the List Interface

VARIABLE/METHOD NAME	VERSION	DESCRIPTION
add()	1.2	Adds an element to the list.
addAll()	1.2	Adds a collection of elements to the list.
clear()	1.2	Clears all elements from the list.
contains()	1.2	Checks to see if an object is a value within the list.
containsAll()	1.2	Checks if the list contains a collection of elements.
equals()	1.2	Checks for equality with another object.
get()	1.2	Returns an element at a specific position.
hashCode()	1.2	Computes hash code for the list.
indexOf()	1.2	Searches for an element within the list.
isEmpty()	1.0	Checks if the list has any elements.
iterator()	1.2	Returns an object from the list that allows all of the list's elements to be visited.
lastIndexOf()	1.2	Searches from the end of the list for an element.
listIterator()	1.2	Returns an object from the list that allows all of the list's elements to be visited sequentially.
remove()	1.2	Clears a specific element from the list.
removeAll()	1.2	Clears a collection of elements from the list.
retainAll()	1.2	Removes all elements from the list not in another collection.
set()	1.2	Changes an element at a specific position within the list.
size()	1.2	Returns the number of elements in the list.
subList()	1.2	Returns a portion of the list.
toArray()	1.2	Returns the elements of the list as array.

You'll find three concrete list implementations in the Collections Framework: `Vector`, `ArrayList`, and `LinkedList`. The `Vector` class from the historical group has been reworked into the framework and was discussed in Chapter 3. `ArrayList` is the replacement for `Vector` in the new framework. It represents an unsynchronized vector where the backing store is a dynamically growable array. On the other hand is `LinkedList`, which, as its name implies, is backed by the familiar linked list data structure.

> **NOTE** *There is no predefined sorted* `List` *implementation (*`TreeList`*). If you need to maintain the elements in a sorted manner, consider either maintaining your elements in a different collection or sorting the elements after you are done inserting (into another collection type). You can manually keep the elements in a* `LinkedList` *in a sorted manner. However, insertion becomes a very costly operation, as you must manually traverse the list before each insertion.*

What's New

Looking quickly at the list doesn't reveal what's added to the `List` interface as compared to the `Collection` interface it extends. Table 9-2 highlights those new methods. You'll find some repeated methods from the `Collection` interface. When also found in the `Collection` interface, the `List` version includes an additional argument to specify position.

Table 9-2. Methods Added to List Interface

VARIABLE/METHOD NAME	VERSION	DESCRIPTION
add()	1.2	Adds an element to the list at a specific index.
addAll()	1.2	Inserts a collection of elements to the list starting at a specific position.
get()	1.2	Returns an element at a specific position.
indexOf()	1.2	Searches for an element within the list.
lastIndexOf()	1.2	Searches from the end of the list for an element.
listIterator()	1.2	Returns an object from the list that allows all of the list's elements to be visited sequentially.
remove()	1.2	Clears a specific element from the list.
set()	1.2	Changes the element at a specific position within the list.
subList()	1.2	Returns a portion of the list.

We'll examine all of the methods of `List` in detail as we look at the specific implementations.

Usage Issues

When using the `List` interface, one of the first problems or at least questions you may run across is the name conflict with the `java.awt.List` class. How can you use both classes in the same program? Or, at a minimum, how can you import both packages into the same program when you need to use only one of the two classes?

- **Solution 1—*Fully qualify everything:***

  The simplest way to use both classes in the same program is to always fully qualify all class names.

  ```
  java.util.List list = new ArrayList();
  java.awt.List listComp = new java.awt.List();
  ```

This gets tedious fast but is less of an issue if you are using the Swing component set and don't ever need to use the `List` component from the Abstract Window Toolkit (AWT).

- **Solution 2—*Only import needed classes:***

  Instead of using wildcard imports, you can import only those classes you need.

  ```
  import java.awt.Graphics;
  import java.util.List;
  ```

  This gets into religious programming issues almost as bad as where to put the `{}`'s on code lines! Personally, I avoid this, as I don't like having a full page of just import lines. If you are using an IDE tool that automatically inserts these, this may be less of an issue.

- **Solution 3—*Double import the specific class you need:***

  Nowadays, with Swing around, you don't use `java.awt.List` any more. If that's the case, just import both packages with a wildcard and import `java.util.List` manually.

  ```
  import java.awt.*;
  import java.util.*;
  import java.util.List;
  ```

  The compiler is smart enough to realize that, if you have `List list;`, you must mean the class that you specifically imported and not the `java.awt.List` class, even though the `java.awt` package was imported first.

> **NOTE** *Regarding Solution 3, this behavior is mandated by Section 7.5 of the Java Language Specification.*

ArrayList Class

The `ArrayList` class is the Collection Framework's replacement for the `Vector` class. Functionally equivalent, their primary difference is that `ArrayList` usage is not synchronized by default, whereas `Vector` is. Both maintain their collection of data in an ordered fashion within an array as their backing store.

> **NOTE** *To clarify terminology,* ordered *means the elements are held according to the positional index inserted, while* sorted *refers to the comparison of element values for ordering.*

The array provides quick, random access to elements at a cost of slower insertion and deletion of those elements not at the end of the list. If you need to frequently add and delete elements from the middle of the list, consider using a `LinkedList` (described later in this chapter).

`ArrayList` shares some similarities with `HashSet` and `TreeSet` and provides some behavior that is not the same. The base implementation class is similar to `HashSet` and `TreeSet`—both extend from the `AbstractCollection` superclass. However, instead of further extending from `AbstractSet`, `ArrayList` extends from `AbstractList`. Unlike the sets, `ArrayList` supports storing duplicate elements. While much of the `ArrayList` behavior is inherited from `AbstractList`, the class still needs to customize the majority of its behavior. Table 9-3 lists the constructors and methods that `ArrayList` overrides.

Table 9-3. Summary of the ArrayList Class

VARIABLE/METHOD NAME	VERSION	DESCRIPTION
ArrayList()	1.2	Constructs an empty list backed by an array.
add()	1.2	Adds an element to the list.
addAll()	1.2	Adds a collection of elements to the list.
clear()	1.2	Clears all elements from the list.
clone()	1.2	Creates a clone of the list.
contains()	1.2	Checks to see if an object is a value within the list.
ensureCapacity()	1.2	Ensures capacity of internal buffer is at least a certain size.
get()	1.2	Returns an element at a specific position.
indexOf()	1.2	Searches for an element within the list.
isEmpty()	1.2	Checks if list has any elements.
lastIndexOf()	1.2	Searches from end of list for an element.
remove()	1.2	Clears a specific element from the list.
removeRange()	1.2	Clears a range of elements from the list.
set()	1.2	Changes an element at a specific position within the list.
size()	1.2	Returns the number of elements in the list.
toArray()	1.2	Returns elements of the list as an array.
trimToSize()	1.2	Trims capacity of internal buffer to actual size.

While Table 9-3 lists (only) sixteen methods, there are actually nineteen versions including overloaded varieties. Add in the eleven inherited methods (excluding the `Object`-specific ones that are not overridden) and there is much you can do with a `List`. We'll look at their usage in groups.

Creating an ArrayList

You can use one of three constructors to create an `ArrayList`. For the first two constructors, an empty array list is created. The initial capacity is ten unless explicitly specified by using the second constructor. When that space becomes too small, the list will increase by approximately half.

```
public ArrayList()
public ArrayList (int initialCapacity)
```

> **NOTE** *Unlike* `Vector`*, you cannot specify a capacity increment. For Sun's reference implementation, the formula to increase capacity is* `newCapacity = (oldCapacity * 3)/2 + 1`*. If you happen to call the constructor with a negative initial capacity, an* `IllegalArgumentException` *will be thrown.*

The final constructor is the copy constructor, creating a new `ArrayList` from another collection:

```
public ArrayList(Collection col)
```

You cannot provide a custom initial capacity. Instead, the internal array will be sized at 10% larger than the collection size. Also, since lists support duplicates, duplicate elements are retained.

One example of creating an `ArrayList` from a list is worth noting. You can create a `List` from an array with `Arrays.asList()`. This is fine if you only want a list that cannot grow. However, if you want to be able to add or remove elements from the list, you must pass that newly created list into a constructor of a concrete collection:

```
String elements[] = {"Schindler's List", "Waiting List", "Shopping List",
  "Wine List"};
List list = new ArrayList(Arrays.asList(elements));
```

Adding Elements

You can add either a single element or a group of elements to the list.

Adding Single Elements

Add a single element to the list by calling the `add()` method:

```
public boolean add(Object element)
public boolean add(int index, Object element)
```

There are two varieties of the `add()` method. When called with only an element argument, the element is added to the end of the list. When `add()` is called with both element and index arguments, the element is added at the specific index and any elements after it are pushed forward in the list. Figure 9-2 should help you visualize this. The program used to add the elements follows.

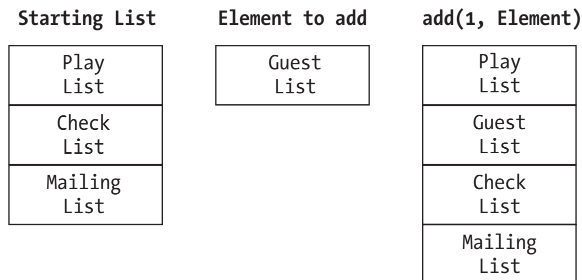

Figure 9-2. Adding elements in the middle of a List.

```
// Create/fill collection
List list = new ArrayList();
list.add("Play List");
list.add("Check List");
list.add("Mailing List");
// Add in middle
list.add(1, "Guest List");
```

> **NOTE** *Like arrays, the index used by a* `List` *starts at zero.*

Since all lists are ordered, the elements are held in the order in which they are added. That is so, unless you explicitly specify the position to add the element, as in `list.add(1, "Guest List")`.

If you are working with a read-only list or one that doesn't support the adding of elements, an `UnsupportedOperationException` will be thrown. To create a read-only list, use the `unmodifiableList()` method of the `Collections` class described in the "Read-Only Collections" section of Chapter 12.

If the internal data structure is too small to handle storing the new element, the internal data structure will automatically grow. If you try to add the element at a position beyond the end of the list (or before the beginning), an `IndexOutOfBoundsException` will be thrown.

Adding Another Collection

You can add a group of elements to a list from another collection with the `addAll()` method:

```
public boolean addAll(Collection c)
public boolean addAll(int index, Collection c)
```

Each element in the collection passed in will be added to the current list via the equivalent of calling the `add()` method on each element. If an index is passed to the method call, elements are added starting at that position, moving existing elements down to fit in the new elements. Otherwise, they are added to the end. The elements are added in the order in which the collection's iterator returns them.

> **NOTE** *Calling* `addAll()` *is efficient in that it will only move the elements of the original list once when adding in the middle of the list.*

If the underlying list changes, true is returned. If no elements are added, false is returned. The only way for false to be returned is if the collection passed in is empty.

If adding elements to the list is not supported, an `UnsupportedOperationException` will be thrown.

Getting an Element

The ArrayList class supports retrieval of a single element by index with get():

```
public Object get(int index)
```

To get the object at a specific position, just call get():

```
Object obj = list.get(5);
```

If you call get() with an index outside the valid range of elements, an IndexOutOfBoundsException is thrown.

Removing Elements

Like sets, lists support removal in four different ways.

Removing All Elements

The simplest removal method, clear(), is one that clears all of the elements from the list:

```
public void clear()
```

There is no return value. However, you can still get an UnsupportedOperationException thrown if the list is read-only.

Removing Single Elements

Use the remove() method if you wish to remove a single element at a time:

```
public boolean remove(Object element)
public Object remove(int index)
```

You can remove a single element by position or by checking for equality. When passing in an element, the equals() method is used to check for equality. As List supports duplicates, the first element in the list that matches the element will be removed.

The value returned depends on which version of remove() you use. When passing in an object, remove() returns true if the object was found and removed

from the list, otherwise, false is returned. For the version accepting an index argument, the object removed from the specified position is returned.

Certain removal-failure cases result in an exception being thrown. If removal is not supported, you'll get an `UnsupportedOperationException` thrown whether the element is present or not. If the index passed in is outside the valid range of elements, an `IndexOutOfBoundsException` is thrown.

Removing Another Collection

The third way to remove elements is with `removeAll()`:

```
public boolean removeAll(Collection c)
```

The `removeAll()` method takes a `Collection` as an argument and removes from the list all instances of each element in the collection passed in. If an element from the passed-in collection is in the list multiple times, all instances are removed.

For instance, if the original list consisted of the following elements:

```
{"Access List", "Name List", "Class List", "Name List", "Dean's List"}
```

and the collection passed in was:

```
{"Name List", "Dean's List"}
```

the resulting list would have every instance of "Name List" and "Dean's List" removed:

```
{"Access List", "Class List"}
```

As with most of the previously shown set methods, `removeAll()` returns true if the underlying set changes and false or an `UnsupportedOperationException`, otherwise. Not surprisingly, the `equals()` method is used to check for element equality.

Retaining Another Collection

The `retainAll()` method works like `removeAll()` but in the opposite direction:

```
public boolean retainAll(Collection c)
```

In other words, only those elements within the collection argument are kept in the original set. Everything else is removed, instead.

Figure 9-3 should help you visualize the difference between `removeAll()` and `retainAll()`. The contents of the starting list are the four lists mentioned previously (Access List, Name List, Class List, Dean's List). The acting collection consists of the elements: Name List, Dean's List, and Word List. The Word List element is included in the acting collection to demonstrate that in neither case will this be added to the original collection.

Starting List	Acting Collection	removeAll()	retainAll()
Access List	Name List	Access List	Name List
Name List	Dean's List	Class List	Dean's List
Class List	Word List		
Dean's List			

Figure 9-3. The removeAll() method versus the retainAll() method.

Removing Ranges

The `removeRange()` method is a protected support method used by `ArrayList`. Unless you subclass `ArrayList`, you'll never use it directly:

```
protected void removeRange(int fromIndex, int toIndex)
```

The indices passed into the method represent the starting index and the index beyond the last element to remove. That means that if `fromIndex` is equal to `toIndex`, nothing will be removed. Also, calling `removeRange()` with an index beyond the ends of the list results in an `ArrayIndexOutOfBoundsException` thrown.

Elements are removed by shifting the remaining elements to fill the space of the removed elements, thus decreasing the size of the list.

List Operations

Lists support several operations that have to do with working with the elements in the collection. There is support for fetching, finding, and copying elements, among some other secondary tasks.

Fetching Elements

To work with all elements of the list, you can call the `iterator()` method to get an `Iterator` or the `listIterator()` method to get a `ListIterator`:

```
public Iterator iterator()
public ListIterator listIterator()
public ListIterator listIterator(int index)
```

Since the elements of a list are ordered, calling `iterator()` or `listIterator()` with no arguments returns the same collection of ordered elements. The only difference is the set of methods you can use with the returned iterator. The second version of `listIterator()` allows you to get an iterator starting with any position within the list.

Using an iterator returned from a `List` is like using any other iterator. The only difference is that the order of the elements in the list is preserved:

```
List list = Arrays.asList(new String[] {"Hit List", "To Do List", "Price List",
  "Top Ten List"});
Iterator iter = list.iterator();
while (iter.hasNext()) {
  System.out.println(iter.next());
}
```

Running this will result in the following output:

```
Hit List
To Do List
Price List
Top Ten List
```

`ListIterator` will be discussed in further detail later in this chapter.

Finding Elements

Besides fetching elements of the list, you can check to see if the list contains a single element or collection of elements.

Checking for Existence

Use the contains() method to check if a specific element is part of the list:

```
public boolean contains(Object element)
```

If found, contains() returns true, if not, false. As with remove(), equality checking is done through the equals() method of the element.

Checking for Position

If, instead of seeing only whether an element is in the list, you want to know where in the list it's located, that's where the indexOf() and lastIndexOf() methods come in:

```
public int indexOf(Object element)
public int lastIndexOf(Object element)
```

Starting at the beginning or end, you can find out where in the vector the next instance of a specific element is located. Unlike Vector, there is no way to start at a position other than the beginning or end.

Both indexOf() and lastIndexOf() use equals() to check for equality and return –1 if the element is not found. Also, the found position is reported from the beginning of the list, not relative to the position searched from.

If you're interested in finding all of the positions for a single element in an ArrayList, you're out of luck. You'll need to convert the list to a Vector and use the versions of indexOf() or lastIndexOf() that support a starting position other than the end.

Checking for List Containment

In addition to checking whether the list includes a specific element, you can check to see if the list contains a whole collection of other elements with the containsAll() method:

```
public boolean containsAll(Collection c)
```

This method takes a Collection as its argument and reports if the elements of the passed-in collection are a subset of the current list. In other words, is each element of the collection also an element of the list? The current list can contain other elements but the passed-in collection cannot or containsAll() will return

false. If an element is in the passed-in collection multiple times, it only needs to be in the source list once to be successful.

Replacing Elements

You can replace elements in a list with the `set()` method:

```
public Object set(int index, Object element)
```

Use the `set()` method when you need to replace the element at a specific position in the list. The object being replaced is returned by the `set()` method. If the index happens to be invalid, an `IndexOutOfBoundsException` is thrown.

Checking Size

To find out how many elements are in a list, use its `size()` method:

```
public int size()
```

`ArrayList` also has an `isEmpty()` method to get the size and to check to see if no elements are in the list:

```
public boolean isEmpty()
```

Checking Capacity

Similar to a `Vector`, an `ArrayList` has a capacity as well as a size. The *capacity* represents the size of the internal backing store, or the number of elements the array list can hold before the internal data structure needs to be resized. Minimizing excess memory usage by the backing store preserves memory but causes a higher insertion-time price when that capacity is exceeded. Working with capacity properly can improve performance immensely without wasting too much memory.

Use the `ensureCapacity()` method to check that the internal data structure has enough capacity before adding elements:

```
public void ensureCapacity(int minimumCapacity)
```

While you don't have to call `ensureCapacity()` yourself, if you plan on inserting a considerable number of elements, it is best to make sure that all of the elements will fit before even adding the first. This will reduce the number of resizes necessary for the internal data structure.

After adding all of the elements you need to add to a list, call the `trimToSize()` method:

```
public void trimToSize()
```

The `trimToSize()` method makes sure that there is no unused space in the internal data structure. Basically, the capacity of the internal data structure is reduced (by creating a new array) if the size is less than the capacity.

Copying and Cloning Lists

`ArrayList` supports the standard mechanisms for duplication. You can clone them, serialize them, copy them, or simply call the standard copy constructor.

The `ArrayList` class is `Cloneable` and has a public `clone()` method:

```
public Object clone()
```

When you call the `clone()` method of an `ArrayList`, a shallow copy of the list is created. The new list refers to the same set of elements as the original. It does not duplicate those elements, too.

Since `clone()` is a method of the `ArrayList` class and not the `List` (or `Collection`) interface, you must call `clone()` on a reference to the class, not the interface:

```
List list = ...
List list2 = ((List)((ArrayList)list).clone());
```

In addition to implementing the `Cloneable` interface, `ArrayList` implements the empty `Serializable` interface. If all of the elements of an `ArrayList` are `Serializable`, you can then serialize the list to an `ObjectOutputStream` and later read it back from an `ObjectInputStream`. The following demonstrates this:

```
FileOutputStream fos = new FileOutputStream("list.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(list);
oos.close();
FileInputStream fis = new FileInputStream("list.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
List anotherList = (List)ois.readObject();
ois.close();
System.out.println(anotherList);
```

Among other uses, `ArrayList`'s serializability is useful when saving information between session requests in a servlet.

ArrayList also supports the standard toArray() methods from Collection for copying elements out of a list into an array:

```
public Object[] toArray()
public Object[] toArray(Object[] a)
```

The first toArray() method returns an Object array containing all the elements in the list. The position of list elements is preserved in the object array. The second version allows you to get an array of a specific type, Object or otherwise. This would allow you to avoid casting whenever you need to get an element out of the array.

WARNING *If the elements of the list are not assignment-compatible with the array type, an* ArrayStoreException *will be thrown.*

Checking for Equality

The ArrayList class gets its equals() method from the AbstractList class:

```
public boolean equals(Object o)
```

An ArrayList is equal to another object if the other object implements the List interface, has the same size(), and contains the same elements in the same positions.

The equals() method of the elements is used to check if two elements at the same position are equivalent.

Hashing Lists

Like equals(), ArrayList gets its hashCode() method from AbstractList:

```
public int hashCode()
```

The hashCode() for a list is defined in the List interface javadoc:

```
hashCode = 1;
Iterator i = list.iterator();
while (i.hasNext()) {
  Object obj = i.next();
  hashCode = 31*hashCode + (obj==null ? 0 : obj.hashCode());
}
```

AbstractList implements the method as such.

LinkedList Class

The final concrete `List` implementation provided with the Collection Framework is the `LinkedList`. The `LinkedList` class is a doubly linked list, which internally maintains references to the previous and next element at each node in the list. As Table 9-4 shows, `LinkedList` has the previously introduced `List` interface methods as well as a special set for working from the ends of the list.

Table 9-4. Summary of the LinkedList Class

VARIABLE/METHOD NAME	VERSION	DESCRIPTION
LinkedList()	1.2	Constructs an empty list backed by a linked list.
add()	1.2	Adds an element to the list.
addAll()	1.2	Adds a collection of elements to the list.
addFirst()	1.2	Adds an element to beginning of the list.
addLast()	1.2	Adds an element to end of the list.
clear()	1.2	Clears all elements from the list.
clone()	1.2	Creates a clone of the list.
contains()	1.2	Checks to see if an object is a value within the list.
get()	1.2	Returns an element at a specific position.
getFirst()	1.2	Returns the first element in a list.
getLast()	1.2	Returns the last element in a list.
indexOf()	1.2	Searches for an element within the list.
lastIndexOf()	1.2	Searches from end of the list for an element.
listIterator()	1.2	Returns an object from the list that allows all of the list's elements to be visited sequentially.
remove()	1.2	Clears a specific element from the list.
removeFirst()	1.2	Removes the first element from the list.
removeLast()	1.2	Removes the last element from the list.
set()	1.2	Changes an element at a specific position within the list.
size()	1.2	Returns the number of elements in the list.
toArray()	1.2	Returns the elements of the list as an array.

As the behavior of most of the `LinkedList` methods duplicates that of `ArrayList`, we'll look only at those that are new or specialized to `LinkedList`.

Creating a LinkedList

There are two constructors for `LinkedList`: one for creating an empty list, and the other, the standard copy constructor:

```
public LinkedList()
public LinkedList(Collection col)
```

Adding Elements

To add a single element, call the add() method:

```
public boolean add(Object element)
public boolean add(int index, Object element)
```

The two basic varieties of add() are the same as ArrayList and add the element to the end of the list unless an index is specified.

You can also treat the linked list as a stack or queue and add elements at the head or tail with addFirst() and addLast(), respectively:

```
public boolean addFirst(Object element)
public boolean addLast(Object element)
```

The addLast() method is functionally equivalent to simply calling add(), with no index.

If you are using a list that doesn't support adding elements, an UnsupportedOperationException gets thrown when any of these methods are called.

Retrieving the Ends

You can use the getFirst() and getLast() methods of LinkedList to get the elements at the ends of the list:

```
public Object getFirst()
public Object getLast()
```

Both methods only *retrieve* the element at the end—they don't remove it. A NoSuchElementException will be thrown by either method if there are no elements in the list.

Removing Elements

LinkedList provides the removeFirst() and removeLast() methods to remove the elements at the ends of the list:

```
public Object removeFirst()
public Object removeLast()
```

While both a stack and a queue usually remove elements from the beginning of a data structure, the `removeLast()` method is provided in case you choose to work with the linked list in reverse order.

Using a list that doesn't support element removal results in an `UnsupportedOperationException` when any of the removal methods are called. Also, calling either method with no elements in the list results in a `NoSuchElementException`.

LinkedList Example

To demonstrate the use of a `LinkedList`, let's create a thread pool. If you aren't familiar with a *thread pool*, it's a collection of threads that sit around waiting for work to be done. Instead of creating a thread when you need work to be done, the creation process happens once and then you reuse the existing threads. You can also limit the number of threads created, preserving system resources.

The pool will maintain a list of `TaskThread` instances that sit around waiting for jobs to be done. Let's look at this class first in Listing 9-1. Basically, the constructor maintains a reference to the thread pool while the thread's `run()` method sits in an infinite loop. The mechanism to get the next job blocks so that the threads aren't sitting around in a busy-wait loop.

Listing 9-1. Defining a thread for a thread pool.

```
public class TaskThread extends Thread {
  private ThreadPool pool;

  public TaskThread(ThreadPool thePool) {
    pool = thePool;
  }

  public void run() {
    while (true) {
      // blocks until job
      Runnable job = pool.getNext();
      try {
        job.run();
      } catch (Exception e) {
        // Ignore exceptions thrown from jobs
        System.err.println("Job exception: " + e);
      }
    }
  }
}
```

The actual ThreadPool class shown in Listing 9-2 is rather simple. When the pool is created, we create all the runnable threads. These are maintained in a LinkedList. We treat the list as a queue, adding at the end and removing at the beginning. When a new job comes in, we add it to the list. When a thread requests a job, we get it from the list. Appropriate synchronization is done to block threads while the list is empty:

Listing 9-2. Defining the thread pool.

```
import java.util.*;

public class ThreadPool {
  private LinkedList tasks = new LinkedList();

  public ThreadPool(int size) {
    for (int i=0; i<size; i++) {
      Thread thread = new TaskThread(this);
      thread.start();
    }
  }

  public void run(Runnable task) {
    synchronized (tasks) {
      tasks.addLast(task);
      tasks.notify();
    }
  }

  public Runnable getNext() {
    Runnable returnVal = null;
    synchronized (tasks) {
      while (tasks.isEmpty()) {
        try {
          tasks.wait();
        } catch (InterruptedException ex) {
          System.err.println("Interrupted");
        }
      }
      returnVal = (Runnable)tasks.removeFirst();
    }
    return returnVal;
  }
```

The test program is included as part of the pool class. It creates a pool of two threads to do work and five jobs that just print a message twenty-five times each:

```
  public static void main (String args[]) {
    final String message[] = {"Reference List", "Christmas List",
      "Wish List", "Priority List", "'A' List"};
    ThreadPool pool = new ThreadPool(message.length/2);
    for (int i=0, n=message.length; i<n; i++) {
      final int innerI = i;
      Runnable runner = new Runnable() {
        public void run() {
          for (int j=0; j<25; j++) {
            System.out.println("j: " + j + ": " + message[innerI]);
          }
        }
      };
      pool.run(runner);
    }
  }
}
```

When you run the program you'll notice that, at most, two threads are running, and as each task finishes, the thread grabs another waiting task from the pool. Part of the output follows:

```
. . .
j: 20: Reference List
j: 21: Reference List
j: 22: Reference List
j: 23: Reference List
j: 24: Reference List
j: 0: Wish List
j: 1: Wish List
j: 2: Wish List
j: 3: Wish List
j: 4: Wish List
j: 5: Wish List
j: 6: Wish List
j: 7: Wish List
j: 0: Priority List
j: 1: Priority List
j: 2: Priority List
. . .
```

The main `ThreadPool` class can be improved in many ways. For instance, you can use lazy initialization to create the threads in the pool, waiting until they are first needed before taking the creation hit. You can also add support for priorities to the queue. Right now, jobs are handled in the order in which they come in. A priority queue implementation is shown in the "Creating Advanced Collections" section of Chapter 16. You can use that as your data structure instead of the `LinkedList` to manage tasks.

ListIterator

The `ListIterator` is an extension to the `Iterator`. Because lists are ordered and support positional access, you can use a `ListIterator` for bidirectional access through the elements of a list. The methods that support this are found in Table 9-5.

Table 9-5. Summary of the ListIterator Interface

VARIABLE/METHOD NAME	VERSION	DESCRIPTION
add()	1.2	Adds an element to the list.
hasNext()	1.2	Checks in the forward direction for more elements in the iterator.
hasPrevious()	1.2	Checks in the reverse direction for more elements in the iterator.
next()	1.2	Fetches the next element of the iterator.
nextIndex()	1.2	Returns the index of the next element of the iterator.
previous()	1.2	Fetches the previous element of the iterator.
previousIndex()	1.2	Returns the index of the previous element of the iterator.
remove()	1.2	Removes an element from the iterator.
set()	1.2	Changes the element at a specific position within the list.

While `Iterator` is very simple to use, there are many different ways to use `ListIterator`.

- **Like an Iterator:**

  Since `ListIterator` extends from `Iterator`, you can use `hasNext()` and `next()` to scroll forward through the list:

```
ListIterator iter = list.listIterator();
while (iter.hasNext()) {
```

```
System.out.println(iter.next());
}
```

- **In reverse order:**

  The `hasPrevious()` and `previous()` methods function similarly to `hasNext()` and `next()`, taking you in the reverse direction. You just need to start at the end:

```
ListIterator iter = list.listIterator(list.size());
while (iter.hasPrevious()) {
  System.out.println(iter.previous());
}
```

- **In both directions:**

  You can intermix `next()` and `previous()` calls to alternate directions.

- **To modify the list:**

  While `Iterator` only supports `remove()`, `ListIterator` adds `set()` and `add()` to the table.

For the last two cases, it is important to understand the concept of cursor position within the iterator. The cursor position is not pointing at an element but is instead positioned between elements (or at an end of the list). It is from this cursor position that all of the operations of the `ListIterator` are performed. For instance, if you have a six-element list, there will be seven cursor positions as shown in Figure 9-4.

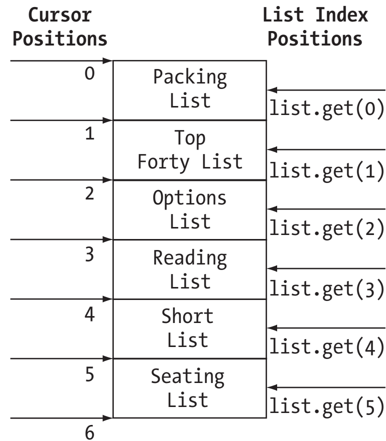

Figure 9-4. Understanding ListIterator cursor positions.

Using the `next()` and `previous()` methods allows you to move the cursor:

```
public Object next()
public Object previous()
```

Using the `nextIndex()` and `previousIndex()` methods allows you to find out the index of the elements around the current cursor position:

```
public int nextIndex()
public int previousIndex()
```

You can then use the index returned to fetch the element from the list if you don't like `next()` and `previous()`. Both methods return –1 if you are at a list end without an element in the appropriate direction: at the end for `nextIndex()` or at the beginning for `previousIndex()`.

Instead of checking for –1 at the ends, you can use `hasNext()` and `hasPrevious()` to detect end cases:

```
public boolean hasNext()
public boolean hasPrevious()
```

Both return true if there are additional elements in the given direction, or false, if not.

The remaining three methods allow you to modify the list that the iterator came from: `add()`, `remove()`, and `set()`.

```
public void add(Object element)
public void remove()
public void set(Object element)
```

Noticeably absent from all three methods is an index. Which element is affected by each call? The `add()` method inserts the new element before the implicit cursor no matter which direction you are traversing through the list. On the other hand, the `remove()` and `set()` methods affect the element last returned by `next()` or `previous()`.

All three operations are optional and would throw an `UnsupportedOperationException` if attempted on a list that didn't support the operations. It is also possible to get an `IllegalStateException` thrown if you try to `set()` or `remove()` an element after `remove()` or `add()` has been called.

To demonstrate using the list from Figure 9-4 as a guide, imagine starting at cursor position 3. Here, `next()` would return "Reading List" and `previous()` would return "Options List." Assuming the last iterator method called was `next()`, you can change the "Reading List" to an "Endangered Species List" with a call to `set("Endangered Species List")`. Or you can remove it with a call to `remove()`. If you were to call `add("Best Sellers List")`, this element would be inserted between "Reading List" and "Options List" no matter which direction you were traversing through the list. When you `add()` an element, the previous and next indices get incremented so the `next()` element would remain the same: "Reading List."

Summary

In this chapter, we explored the many facets of the `List` interface and its two new concrete implementations: `ArrayList` and `LinkedList`. With `ArrayList`, you have quick, random access, while `LinkedList` provides quick insertions and deletions at positions other than the end. Which of the two you use depends upon your specific needs but you'll use both frequently instead of the historical `Vector` class to maintain ordered lists of elements. You also learned how to take advantage of the `ListIterator` interface to traverse a list bidirectionally (instead of just forward as with an `Iterator`).

The next chapter introduces the `Map` interface with its `HashMap` and `TreeMap` implementations. You'll learn to use these classes to maintain key-value pairs, either unordered or ordered. You'll also learn about the `WeakHashMap` and how you can use it effectively to maintain a weakly referenced map.