8 Collections

This chapter covers

- New methods in the Collections class
- LinkedHashMap and LinkedHashSet
- IdentityHashMap
- The RandomAccess interface

The release of JDK 1.4 adds a few minor features to the Collections Framework in the `java.util` package. A number of utility methods have been added to the `Collections` class; each of these deals with `Lists` in some way.

There are three new classes discussed in this chapter: `LinkedHashMap`, `LinkedHashSet`, and `IdentityHashMap`. The first two are variations on the `HashMap` and `HashSet` classes (respectively), and they preserve the order in which objects are added to them. The third, `IdentityHashMap`, is a `Map` that overrides the `hashCode()` methods of its keys, allowing objects to be differentiated based on identity rather than on content.

There is also a new marker interface*, `RandomAccess`, which allows a class to declare that it is suitable for fast random access. More precisely, this means that it has efficient implementations of the `get()` and `set()` methods.

8.1 Utilities

A number of utility methods have been added as static methods in the `Collections` class. These utilities provide a number of simple but commonly used routines. If you've used the Collections Framework a lot, you've probably written some of these utilities yourself; now they are available as part of the library.

For some of these, the documentation claims that the implementation is faster than "the naive implementation." Being part of the Collections Framework probably helps these methods use private information to make themselves faster. As a result, these routines should be preferred over hand-rolled implementations.

8.1.1 Rotating list elements

The `Collections.rotate()` method rotates the elements of a list, which means that it advances each element a certain number of steps. Elements that are advanced off the end of the list are wrapped around to the beginning of the list, which is why this is called `rotate()` and not `shift()`. Rotation can go in the negative direction as well, which means that each element can be moved backward, and elements that fall off the front of the list are wrapped around to the end.

This is the type signature of the `rotate()` method:

```
Collections.rotate( List list, int distance );
```

The `distance` value specifies the number of steps to rotate. This value can be negative.

*A *marker interface* is an interface that doesn't contain any methods; a class implements such an interface merely to signal that it has a certain property.

Figure 8.1 shows a rotation of 1. Each element in the list is moved forward by one step, as shown by the arrows on top. The element at the end cannot be moved forward, so it wraps around to the front of the list, occupying the first slot in the list.

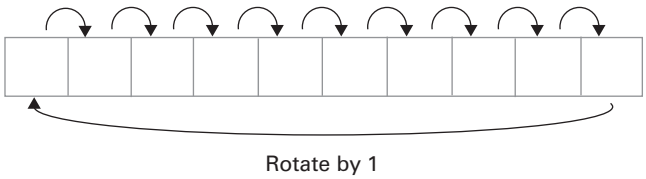

Figure 8.1 A rotation by 1. Each element is moved forward by one step. The last element of the list must wrap around to the beginning of the list.

It is also possible to rotate a sublist of a larger list. The elements of the sublist are *aliases* for the elements of the containing list, which means that changes made to the sublist are reflected in the containing list. Thus, a rotation of the elements of a sublist affects the selected elements of the containing list. This can be very useful, since it allows you to rotate an arbitrary contiguous portion of a list.

As an example, we'll use a list of 10 elements, and we'll choose from that list a sublist of five elements, ranging from element two to element six, inclusive. This sublist is taken from the larger list using the `subList()` method (see figure 8.2).

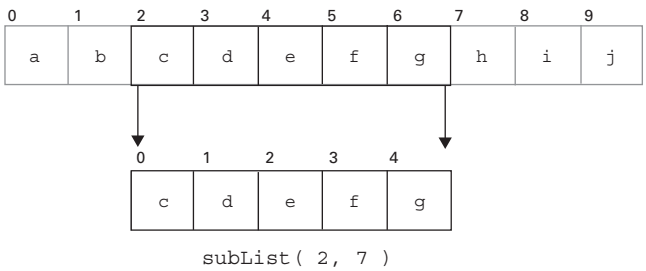

Figure 8.2 A sublist of a larger list. This sublist is created by a call to the `subList()` method. The elements of this sublist are aliases for the elements of the containing list, so any modifications that we make to the sublist are reflected in the containing list.

By calling `rotate()` on this sublist, we rotate the elements of the sublist as if the sublist were its own list. Elements that are in the larger list, but are outside the sublist, are not affected (see figure 8.3).

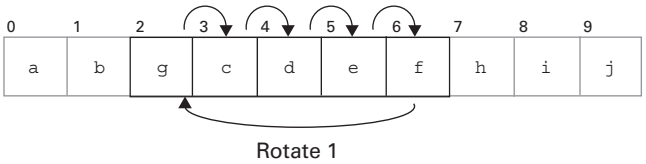

Figure 8.3 A rotation of the elements of a sublist. Each element in the sublist is moved forward by one; the final element wraps around to the front of the sublist. If we confine our view to the sublist, this is a normal rotation.

8.1.2 Replacing list elements

The `Collections.replaceAll()` method replaces each occurrence of a particular element with another element. The signature of this method looks like this:

```
Collections.replaceAll( List list, Object oldVal, Object newVal );
```

Each occurrence of the value `oldVal` within the list, `list`, is replaced with `newVal`.

8.1.3 Finding sublists within lists

The `Collections.indexOfSubList()` method finds the first occurrence of a particular sublist within a containing list. It's rather like the `indexOf()` method, but it takes a `List`, rather than an `Object`, as the target to look for:

```
int Collections.indexOfSubList( List source, List target );
```

The value returned is the index, within the containing array, of the first element of the target list.

For example, let's look for the list { 1, 2 } within a larger list of random-looking digits. This list might occur multiple times, but `indexOfSubList()` only returns the first occurrence (see figure 8.4). In this example, the sublist { 1, 2 } occurs three

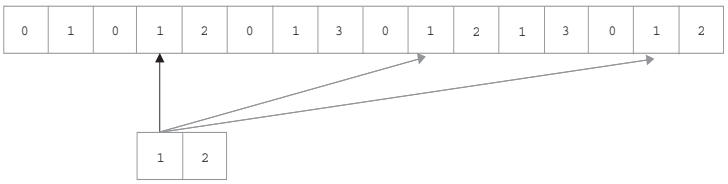

Figure 8.4 A sublist might be found multiple times within the containing list. `indexOfSubList()` returns the *first* of these occurrences.

times within the containing list, at offsets 3, 9, and 14, but `indexOfSubList()` only returns the value 3.

`Collections.lastIndexOfSubList()` finds the *last* occurrence of the target list within the containing list (see figure 8.5). In this example, `lastIndexOfSubList()` would return 14.

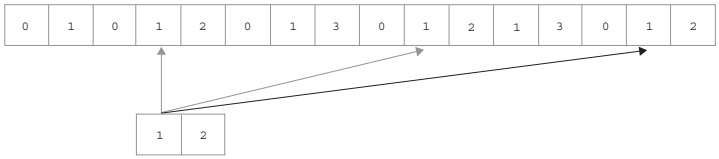

Figure 8.5 `lastIndexOfSubList()` returns the index of the *last* occurrence of the sublist within the containing list.

Both methods return `-1` if the target list is not found within the containing list.

8.1.4 Swapping list elements

The `Collections.swap()` utility method swaps two elements of a list. These elements are specified by two integers, which serve as indices into the list. Figure 8.6 shows a swap of elements `c` and `g` within a 10-element list:

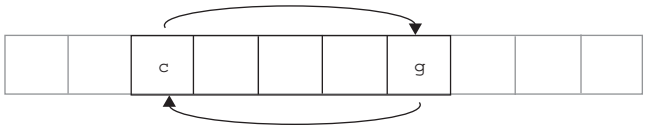

Figure 8.6 `swap()` swaps the position of elements `c` and `g` within a 10-element list.

In code, it looks like this:

```
Collections.swap( list, 2, 6 );
```

8.1.5 Converting enumerations to lists

The `Collections.list()` method converts an `Enumeration` to a `List`. The signature of this method is as follows:

```
Collections.list( Enumeration e );
```

`list()` achieves this conversion by running through all the elements of the `Enumeration` and adding them onto a list. The following is the equivalent of this method:

```
Enumeration e;

List list = new ArrayList();
while (e.hasMoreElements()) {
  list.add( e.nextElement() );
}
```

Most `Enumeration`s are created from arrays or lists that are contained entirely within RAM. However, it is possible to create an `Enumeration` that "contains" more objects than could possibly fit in RAM. This is possible because it is not required that all of the elements of an `Enumeration` be in RAM at the same time—only that they be available, one at a time.

It's even possible to create an `Enumeration` that never ends. The `Enumeration` in listing 8.1 returns the prime numbers in order, starting at 2.

Listing 8.1 EndlessEnumeration.java

(See \Chapter8\EndlessEnumeration.java)

```
import java.util.*;

public class EndlessEnumeration implements Enumeration
{
  // The last prime we returned
  // initialize it to be before the first prime
  private int lastPrime = 1;

  // There are always more primes
  public boolean hasMoreElements() {
    return true;
  }

  public Object nextElement() {
    // Start searching from after the last one we found
    int n = lastPrime+1;
    while (true) {
      if (isPrime( n )) {
        lastPrime = n;
        return new Integer( n );
      } else {
        n++;
      }
    }
  }

  private boolean isPrime( int n ) {
    // 2 is the lowest possible factor
    // n/2 is the highest possible factor
    for (int i=2; i<=n/2; ++i) {
      // If n is divisible by i, then it's not prime
      if ((n%i)==0) {
```

Check each integer until we find the next prime

Divide by every possible factor to find out if it's prime

```
        return false;
      }
    }
    return true;
  }

  static public void main( String args[] ) throws Exception {
    Enumeration e = new EndlessEnumeration();
    for (int i=0; i<20; ++i)
      System.out.println( e.nextElement() );
  }
}
```

The `hasMoreElements()` method always returns `true`, because this `Enumeration` always has more elements. If you pass an object of this class to `Collections.list()`, it will just run and run and run, generating primes endlessly, never returning from the call to `Collections.list()`. (Since the prime generation algorithm is pretty slow, the machine will probably fail before it runs out of memory to store the primes.)

8.2 LinkedHashMap and LinkedHashSet

The `LinkedHashMap` and `LinkedHashSet` classes are much like their unlinked counterparts, `HashMap` and `HashSet`, respectively. They differ in that each of these classes remembers the order in which entries were inserted. Iterating over the elements of these collections will produce the elements in the same order in which they were inserted.

8.2.1 Using LinkedHashMap

As an example, we'll create a `LinkedHashMap` as shown in figure 8.7 using the following code:

```
LinkedHashMap lhm = new LinkedHashMap();
lhm.put( "a", "Albert" );
lhm.put( "b", "Barbara" );
lhm.put( "c", "Chuck" );
```

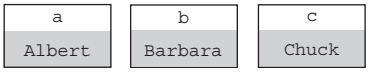

Figure 8.7 A mapping from single-character strings to names

We can iterate through the entries of this mapping as follows:

```
for (Iterator it = lhm.entrySet().iterator(); it.hasNext();) {
  Map.Entry me = (Map.Entry)it.next();
  System.out.println( me.getKey()+" --> "+me.getValue() );
}
```

The preceding code produces the following output:

```
a --> Albert
b --> Barbara
c --> Chuck
```

Since we used a `LinkedHashMap`, the entries of the map come out in the same order in which they were inserted. This is because a set of links are maintained inside the `LinkedHashMap` that form a kind of hidden linked list (as shown in figure 8.8).*

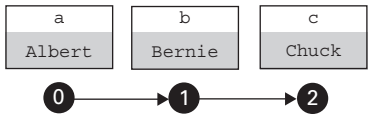

Figure 8.8 The mapping is augmented by a hidden linked list, which specifies an order of the entries.

Inserting vs. updating

Note that the `LinkedHashMap` maintains the order in which the entries were *inserted*. If the entry is *updated*, the ordering doesn't change. (In this context, we define an *update* as a call to `put()`, which uses a key that already exists in the mapping.) For example, we'll overwrite the value `Barbara` with the value `Bernie`:

```
LinkedHashMap lhm = new LinkedHashMap();
lhm.put( "a", "Albert" );
lhm.put( "b", "Barbara" );
lhm.put( "c", "Chuck" );
lhm.put( "b", "Bernie" );
```

The output of the preceding code shows that the value of the second entry has changed, but the order is the same as it was before:

```
a --> Albert
b --> Bernie
c --> Chuck
```

* Actually, it is a doubly linked list, which allows traversal to proceed in either direction.

Once an entry has been inserted into the hidden linked list, it is not moved, as shown in figure 8.9.

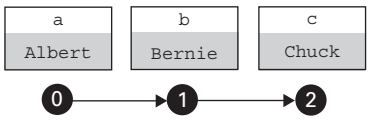

Figure 8.9 Putting a new value into a slot that already exists doesn't change the ordering of the entries. Here, the value of `Barbara` has been replaced with `Bernie`, but the ordering is the same.

Reinserting

In a `LinkedHashMap`, removing an old value and inserting a new value in its place has a different effect on the ordering than simply overwriting the old value with the new value. If you remove the value in question before reinserting it, as we do in the following code fragment, the ordering is changed:

```
LinkedHashMap lhm = new LinkedHashMap();
lhm.put( "a", "Albert" );
lhm.put( "b", "Barbara" );
lhm.put( "c", "Chuck" );
lhm.remove( "b" );
lhm.put( "b", "Bernie" );
```

The output shows that the ordering *has* changed this time:

```
a --> Albert
c --> Chuck
b --> Bernie
```

What is different about this case? The removal doesn't just remove the entry from the mapping; it removes it from the hidden linked list. The removed entry had a fixed position within the ordering defined by the hidden linked list, but that position is lost when the entry is removed (see figure 8.10). Adding the entry back into the `LinkedHashMap` (albeit with a different value) causes it to be put at the end of

Figure 8.10 Removing an entry from the `LinkedHashMap` also removes it from the hidden linked list. Whatever position the removed entry may have had in the ordering of the entries is lost.

the hidden linked list. In this sense, it is as if the entry has been added for the first time (see figure 8.11).

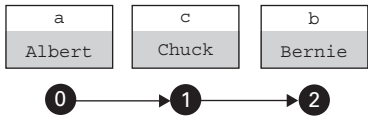

Figure 8.11 Putting the entry back in after removing it causes it to be put at the end of the hidden linked list.

Removing any element from the `LinkedHashMap` removes any information about its position within the insertion order.

8.2.2 Using LinkedHashSet

A `LinkedHashSet` is rather like a `LinkedHashMap`, except that it is a `Set` rather than a `Map`. Like a `LinkedHashMap`, a `LinkedHashSet` remembers the order in which objects are inserted. When you iterate over the objects, the elements of the set are produced in the same order in which they were inserted.

The following code illustrates this:

```
LinkedHashSet lhs = new LinkedHashSet();
lhs.add( "Albert" );
lhs.add( "Barbara" );
lhs.add( "Chuck" );
```

Iterating over the elements goes like this:

```
for (Iterator it = lhs.iterator(); it.hasNext();) {
  System.out.println( it.next() );
}
```

The preceding code produces the following output:

```
Albert
Barbara
Chuck
```

The insertion order is maintained using a linked list, as shown in figure 8.12.

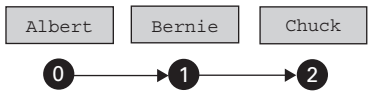

Figure 8.12 The set is augmented by a hidden linked list that specifies an order of the entries.

Adding a value more than once doesn't change the set, and it doesn't change the ordering either:

```
LinkedHashSet lhs = new LinkedHashSet();
lhs.add( "Albert" );
lhs.add( "Barbara" );
lhs.add( "Chuck" );
lhs.add( "Barbara" );
lhs.add( "Albert" );
```

The preceding code produces identical output to the earlier code, with the elements in the same order as before:

```
Albert
Barbara
Chuck
```

However, like the `LinkedHashMap`, removing and reinserting an element does change the order:

```
LinkedHashSet lhs = new LinkedHashSet();
lhs.add( "Albert" );
lhs.add( "Barbara" );
lhs.add( "Chuck" );
lhs.remove( "Barbara" );
lhs.add( "Barbara" );
```

Because `Barbara` was removed and reinserted in the preceding code, it moves to the end of the list:

```
Albert
Chuck
Barbara
```

This is because insertion-order information isn't maintained for elements that have been removed (see figure 8.13).

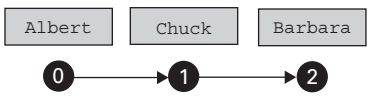

Figure 8.13 After removing `Barbara` and putting it back, it is put at the end of the hidden linked list. Ordering information is not maintained for removed elements.

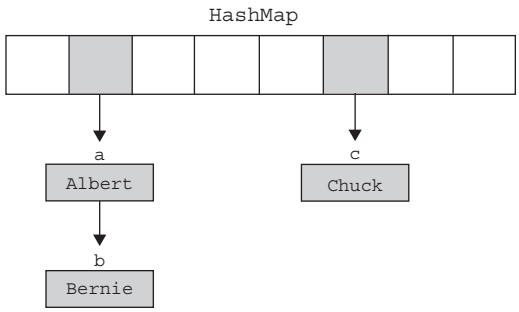

Figure 8.14 `HashMap` and `LinkedHashMap` are both implemented as an array of linked lists. Individual elements are stored in the linked lists; the head of each list occupies one of the slots of the array. The arrays are *sparse*, meaning that a significant number of the slots are empty.

8.2.3 Efficiency of LinkedHashMap and LinkedHashSet

`LinkedHashMap`s and `LinkedHashSet`s have an efficiency roughly equal to that of `HashMap`s and `HashSet`s, respectively. The operation of adding or removing an element is slightly slower, since the hidden linked lists must be maintained.

Strangely enough, however, iterating over the elements of a `LinkedHashMap` or `LinkedHashSet` is *faster* than it is for the unlinked varieties. To understand this, it is necessary to understand the way these classes are implemented. Since `LinkedHashSet` is implemented using a `LinkedHashMap`, we will not need to discuss it separately.

`LinkedHashMap` and `HashMap` are both implemented as an array of linked lists. Each array slot corresponds to a *hash bin*. A full discussion of hashing and hashing techniques is beyond the scope of this chapter; suffice it to say that, internally, these two classes are represented as an array of linked lists (see figure 8.14).

The arrays are *sparse*, which means that a significant number of the slots are empty. (The average, based on the default settings, is one-quarter empty.) Traversing this array, then, incurs a certain waste of time, since the traversal must visit each slot in the array whether it is empty or not (see figure 8.15).

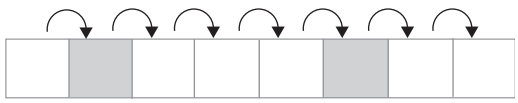

Figure 8.15 Traversing the elements of a `HashMap` requires traversing the entire array, which includes a significant number of elements that are empty.

A `LinkedHashMap`, on the other hand, has the added benefit of another linked list forming yet another chain through the elements. Traversing this list doesn't require traversing empty slots, because there are no empty slots in the linked list (see figure 8.16).

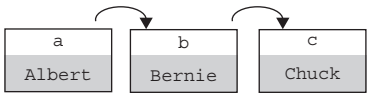

Figure 8.16 Traversing the elements of a `LinkedHashMap` only requires traversing the elements themselves. There are no empty slots in the linked list to waste time on.

It's important to understand that this extra efficiency doesn't come for free. It's the very act of maintaining the linked list that provides this speed boost during iteration, and this maintenance adds a bit of overhead to every operation that adds or removes an entry. A `LinkedHashMap` (or `LinkedHashSet`) might be faster for your application, but it might not—it all depends on how much time is spent building and modifying the hash table, and how much time is spent traversing it. A `LinkedHashMap` entry also uses a bit more memory.

8.2.4 Example: searching a file path

A file path—such as the Java classpath—is a perfect example of an ordered collection. It consists of a list of directories with a fixed order. To look for a file in a path means to look for a file in that collection of directories, respecting the order of the directories in the path.

As an example, we'll create a `Path` object that represents such a file path. You build a `Path` object by adding a series of directories to it, and then call its `findFile()` method to locate a file somewhere in the path.

This `Path` object also allows you to make a directory inactive in path—and make it active again—using the `setActive()` method. To this end, each directory in the `Path` has a *name* that can be used as a handle for making that directory active or inactive. This can be very useful if you are developing multiple projects with multiple tools. You might need to use different versions of a compiler for different projects (or for different parts of the same project!). Using a path-generation system can be a lot easier than editing configuration scripts and restarting your shell over and over.

The `Path` class also includes a `formatAsPath()` method, which formats the path in traditional classpath format.

The following is the listing of a test program that uses the `Path` class. It assumes the existence of two directories, "a" and "b", each of which contains a file called Test.java as well as some other .java files:

```
0 drwxr-xr-x   4 mito     100             0 Dec  3 16:38 .
0 drwxr-xr-x   2 mito     100             0 Dec  3 16:39 ./a
1 -rw-r--r--   1 mito     100            10 Dec  3 16:39 ./a/Test.java
1 -rw-r--r--   1 mito     100            10 Dec  3 16:51 ./a/Foo.java
0 drwxr-xr-x   2 mito     100             0 Dec  3 16:39 ./b
1 -rw-r--r--   1 mito     100            10 Dec  3 16:39 ./b/Test.java
1 -rw-r--r--   1 mito     100            10 Dec  3 16:51 ./b/Bar.java
```

We will create `Path` objects to search for instances of Test.java in these directories. The program is shown in listing 8.2, and it is interleaved with the output it produces, shown in bold italic.

Listing 8.2 PathTest.java

(See \Chapter8\PathTest.java)

```
public class PathTest
{
  static public void main( String args[] ) {
    Path path = null;

    path = new Path();
    path.add( "a", ".\\pathtest\\a\\" );
    path.add( "b", ".\\pathtest\\b\\" );
    System.out.println( path.formatAsPath() );

// .\pathtest\a;.\pathtest\b          (1) Directory "a" followed by directory "b"

    path = new Path();
    path.add( "b", ".\\pathtest\\b\\" );
    path.add( "a", ".\\pathtest\\a\\" );
    System.out.println( path.formatAsPath() );

// .\pathtest\b;.\pathtest\a          (2) Directory "b" followed by directory "a"

    path = new Path();
    path.add( "a", ".\\pathtest\\a\\" );
    path.add( "b", ".\\pathtest\\b\\" );
    System.out.println( path.findFile( "Test.java" ) );

// .\pathtest\a\Test.java             (3) Test.java found in directory "a"

    path.setActive( "a", false );
    System.out.println( path.findFile( "Test.java" ) );

// .\pathtest\b\Test.java             (4) Test.java found in directory "b" because
                                       directory "a" was made inactive
  }
}
```

1 2 Note that the order of the directories of a path matches precisely the order in which the directories were added to the `Path` object.

3 Both "a" and "b" contain a file called Test.java. Since the search looks in "a" before "b", it finds the a\Test.java file first.

4 The same path was searched as in #3, except that, this time, directory "a" was deactivated. As a result, b\Test.java is found.

The source for `Path` is shown in listing 8.3.

Listing 8.3 Path.java

(See \Chapter8\Path.java)

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

public class Path
{
  private LinkedHashMap directories = new LinkedHashMap();

  public Path() {
  }

  public void add( String name, String directory ) {
    add( name, new File( directory ) );
  }

  public void add( String name, File directory ) {
    Entry entry = new Entry( directory );
    directories.put( name, entry );
  }

  public void remove( String name ) {
    directories.remove( name );
  }

  public void setActive( String name, boolean active ) {
    Entry entry = (Entry)directories.get( name );
    if (entry == null)
      throw new NoSuchElementException(
        "No element "+name+" in "+this );
    entry.active( active );
  }

  public File findFile( String target ) {
    final File targetFile = new File( target );
    FileFilter filter = new FileFilter() {
      public boolean accept( File pathname ) {
        // This filter accepts files matching the target file
        return targetFile.getName().equals( pathname.getName() );
```

1 **Store the ordered set of directories**

2 **Add a new directory to the path**

3 **Activate or deactivate a directory**

4 **Find a file somewhere in the path**

```
      }
    }; // end of anonymous inner class

    // Check each directory of the path in turn
    for (Iterator it=directories.keySet().iterator();
         it.hasNext();) {
      String name =  (String)it.next();
      Entry entry = (Entry)directories.get( name );

      // If this directory has been de-activated,
      // don't look in it
      if (!entry.active())
        continue;

// Search the directory with the filter
      File dir = entry.directory();
      File files[] = dir.listFiles( filter );

      if (files != null && files.length>0) {
        // listFiles() should only return one file
        return files[0];
      }
    }
    return null;
  }

  public String formatAsPath() {
    String s = "";
    for (Iterator it=directories.keySet().iterator();
         it.hasNext();) {
      String name =  (String)it.next();
      Entry entry = (Entry)directories.get( name );
      File dir = entry.directory();
      s += dir;
      if (it.hasNext()) {
        s += File.pathSeparator;
      }
    }
    return s;
  }
}

class Entry
{
  private File directory;
  private boolean active;

  public Entry( String name ) {
    this( new File( name ) );
  }

  public Entry( File directory ) {
    this.directory = directory;
    active = true;
```

```
  }

  public boolean active() {
    return active;
  }

  public void active( boolean active ) {
    this.active = active;
  }

  public File directory() {
    return directory;
  }
}
```

1. `LinkedHashMap` has two axes of organization: the hash axis, on which we map directory names to directories; and the order axis, on which we store the ordering of the elements. The ordering matches the order in which the elements were added to the `Path`.
2. `add()` and `remove()` simply call the respective `add()` and `remove()` methods of the `directories` `LinkedHashMap`, and so the order in which directories are added is preserved.
3. `setActive()` lets you selectively turn various directories of the path on and off. This is easier than adding and removing them, because adding and removing have the side effect of changing the order of the directories. `setActive()` can disable a directory without changing its location in the overall order.
4. `findFile()` traverses the list of directories in order, looking in each one to see if the file in question is there. It uses a `FileFilter` object to search each directory. If `findFile()` finds a file in a directory, it returns it immediately.

Note that we needed a `LinkedHashMap` for this program because the `Path` object needed to satisfy the following requirements:

- Sequential access—The directories must be traversed in a particular order
- Random access—It must be possible to activate and deactivate the directories individually

If we only had the first requirement, a `List` would have been sufficient. If we only had the second requirement, a `HashMap` would have been sufficient. However, to satisfy both, we need a `LinkedHashMap`.

8.3 IdentityHashMap

A regular `HashMap` considers two objects to be equal if

```
object0.equals( object 1 )
```

An `IdentityHashMap`, on the other hand, doesn't use the `equals()` method. It only considers two objects to be equal if

```
object0 == object1
```

This alteration in the behavior of the hashing method has subtle but important effects on the way `IdentityHashMap` behaves.

8.3.1 Object equality

Every Java object has a method called `equals()`, which is used to compare objects. The default implementation from `Object` compares objects using their identities, which means that two objects are equal if and only if they have the same reference. `Object.equals()` is implemented as follows:

```
public boolean equals( Object obj ) {
  return (this == obj);
}
```

Many objects override the `equals()` method because they wish to have a definition of equality that is based on the semantics of the object rather than its reference. For example, two separate `Integer` object references are equal if they have the same value, even if they are separate objects:

```
Integer i0 = new Integer( 40 );
Integer i1 = new Integer( 40 );
System.out.println( "i0 == i1: "+(i0==i1) );
System.out.println( "i0.equals( i1 ): "+i0.equals( i1 ) );
```

The preceding code results in the following output:

```
i0 == i1: false
i0.equals( i1 ): true
```

Clearly, these are separate objects, since they are created separately by two different calls to `new Integer()` and thus have different references. This is reflected in the fact that the `==` operator returns `false`. However, the `equals()` method returns `true` because it overrides the implementation in `Object` with another version that only returns `true` if the two `Integer` objects have the same value, regardless of whether they are the same object or not.

8.3.2 Hashing and equality

The exact implementation of equality is very important to the behavior of a hash table. When a key/value pair is inserted for a key that already exists in the table, the new value replaces the old value. Hash tables use the `equals()` method to determine if the key is already in the table.

Continuing to use `Integer` as an example, let's take a look at what happens when we insert two pairs that use the same exact `Integer` object:

```
Integer i0 = new Integer( 40 );

HashMap hm = new HashMap();
hm.put( i0, "first" );
hm.put( i0, "second" );

for (Iterator it = hm.entrySet().iterator(); it.hasNext();) {
  Map.Entry me = (Map.Entry)it.next();
  System.out.println( me.getKey()+" --> "+me.getValue() );
}
```

In this example, a key is used for two different values. Because the exact same object is used, the second value overwrites the first value, and so there is only a single value in the `HashMap` afterwards:

```
40 --> second
```

`IdentityHashMap` can help you avoid this overwriting. It treats two distinct `Integer` objects as different objects, even if they both contain the same integer value.

In table 8.1, we've used both `HashMap` and `IdentityHashMap` to try this out. For each class, we've tried two variations: one where we use the same `Integer` object as the key for both values, and one where we use two different `Integer` objects as keys. As you can see from the results in table 8.1, the only case in which the keys are treated as different objects is the last one, where we used an `IdentityHashMap`, and where we used two different `Integer` objects as keys.

Don't be fooled by the fact that these keys *look* identical. It's not possible for a `HashMap` to contain two different pairs with the same key. These two keys are `Integers` that have the same value and that have the same printed representation (as generated by their `toString()` methods), but they are separate objects and are treated as such by the `IdentityHashMap`.

8.3.3 Example: using the IdentityHashMap

Technically speaking, the `IdentityHashMap` violates one of the general principles of the `Map` interface, which is that `equals()` is always used to compare objects. `IdentityHashMap` should not be used frivolously, but there are situations when it is definitely called for.

Table 8.1 Comparison of `HashMap` and `IdentityHashMap`. They behave differently when given two separate `Integer` objects that have the same value—`IdentityHashMap` treats them as separate objects, while `HashMap` does not.

Hash table	Code	Results

`HashMap`	`Integer i0 = new Integer( 40 );`
	`hm.put( i0, "first" );`
	`hm.put( i0, "second" );`
		`40 --> second`

`HashMap`	`Integer i0 = new Integer( 40 );`
	`Integer i1 = new Integer( 40 );`
	`hm.put( i0, "first" );`
	`hm.put( i1, "second" );`
		`40 --> second`

`IdentityHashMap`	`Integer i0 = new Integer( 40 );`
	`Integer i1 = new Integer( 40 );`
	`ihm.put( i0, "first" );`
	`ihm.put( i0, "second" );`
		`40 --> second`

`IdentityHashMap`	`Integer i0 = new Integer( 40 );`
	`Integer i1 = new Integer( 40 );`
	`ihm.put( i0, "first" );`
	`ihm.put( i1, "second" );`
		`40 --> first`
		`40 --> second`

The following program, `DumpableGraph` (see listing 8.4), contains a simple implementation of a graph. It contains an inner class called `Node`. Each node has a *content string* that can be thought of as the name of the node, and each node also has a set of children. A node has a value and zero or more children. If a node has no children, it's a leaf; otherwise, it's a branch. A graph is one or more nodes connected to each other by the parent-child relation. (We would call it a tree, but it's really a graph because we are permitting back-references and cycles.)

`DumpableGraph` is dumpable because it has a method called `dump()`, which traverses the graph, printing out its contents. In this implementation, it is possible for a node to be its own ancestor, and so it is possible that the graph will contain a cycle. As a result, the `dump()` method has to be careful not to get caught in a loop as it traverses the graph; to do this, it uses a `Map` to keep track of the nodes that it has already dumped, thus avoiding an infinite loop.

Initially, we'll use a regular `HashMap` to keep track of the nodes, and we'll find that it doesn't work correctly. The bug is caused by the fact that our graph can have distinct nodes with the same content string. Once we've visited a node with the string "a", we won't visit any other nodes that also have the string "a", because we'll assume that these are the same node, even though they might not be. As we'll see, the solution is to use an `IdentityHashMap` instead of a regular `HashMap`.

But first, the flawed implementation is shown in listing 8.4.

Listing 8.4 DumpableGraph.java

(see \Chapter8\DumpableGraph.java)

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

public class DumpableGraph
{
  public static class Node {          ● Inner class Node
    private Object obj;
    private List children = new ArrayList();

    public Node( Object obj ) {
      this.obj = obj;
    }

    public void addChild( Node node ) {
      children.add( node );
    }

    public boolean equals( Object node ) {
      return obj.equals( ((Node)node).obj );
    }                                 ● Implement custom
                                        equality/hashing
    public int hashCode() {
      return obj.hashCode();
    }

    public String toString() {
      return obj.toString();
    }

    public void dump() {
      // Start the dumping process with an empty
      // seen-set
      dump( "", new HashMap() );      ● Use a regular Map
    }

    private void dump( String prefix, Map seen ) {   ● dump() traverses
      // Print out information about this node           the node graph...
      System.out.println( prefix+"Node: "+obj+
                          " ["+System.identityHashCode( obj )+"/"+
                          obj.hashCode()+"]" );
      if (children.size()==0) {

        // If there are no children, we've reached a leaf,
        // so we're done
        System.out.println( prefix+"  (no children)" );
      } else {

        // We only visit the children of this node if we
        // haven't already done so -- if we're not in the
```

```
        // seen-set
        if (!seen.containsKey( this )) {                    ...taking care not to visit a node twice

          // Remember that we've processed this node by
          // putting it in the seen-set
          seen.put( this, null );

          // Dump all the children of this node
          for (Iterator it=children.iterator(); it.hasNext();) {
            Node node = (Node)it.next();

            // Indent the prefix by two spaces
            node.dump( prefix+"  ", seen );
          }
        } else {
          System.out.println( prefix+"  (loop)" );
        }
      }
    }
  }

  static public void main( String args[] ) {    Build the graph
    Node a = new Node( "a" );
    Node b = new Node( "b" );
    Node c = new Node( "c" );
    Node d = new Node( "d" );
    Node a2 = new Node( "a" );
    Node b2 = new Node( "b" );
    Node e = new Node( "e" );
    a.addChild( b );
    a.addChild( c );
    c.addChild( d );
    c.addChild( a );
    c.addChild( a2 );
    a2.addChild( b2 );
    a2.addChild( e );
    a.dump();
  }
}
```

Let's take a look at the program in action. Figure 8.17 shows an example graph. Each node is represented by a letter in a black circle, and the arrows point from parents to their children. The variable from the `main()` method in Listing 8.4 that holds the `Node` object is next to the black circle. Note that there are two nodes that have the same content string—nodes "a" and "a2" both have the content string "a".

When we run DumpableGraph, the output shows that we have a problem:

```
Node: a [9751148/97]
  Node: b [7947172/98]
    (no children)
```

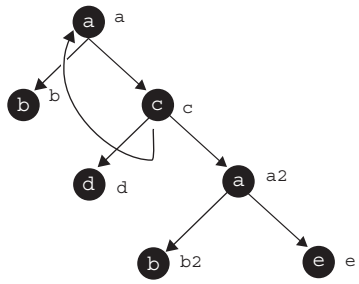

Figure 8.17 A graph composed of `DumpableGraph.Node` objects. Each node has a *content string*, shown inside the circle. The variable names next to the circles show the variables from listing 8.4 that contain the nodes. Note that node "c" has two children that are completely different `Node` objects, but which have the same content string of "a".

```
Node: c [4719703/99]
  Node: d [1375836/100]
    (no children)
  Node: a [9751148/97]
    (loop)
  Node: a [9751148/97]
    (loop)
```

The first boldfaced line shows that the top node "a" is a child of node "c". This corresponds to the curved arrow in figure 8.17. The `dump()` routine stops here because we don't want to get caught in a loop. But the second boldfaced line shows that there's another node called "a" that is a child of "c"—this is the node marked "a2" in the figure. This node has not yet been visited, and yet our `dump()` routine gives up and refuses to follow the arrow to that node.

The problem here is that we have two nodes with the content string "a". What's more, our `dump()` routine uses a regular old `HashMap` to keep track of the nodes we've already seen. As far as `HashMap` is concerned, these two nodes with the same content string *are the same node*. The `equals()` method would say that they are the same, and the `hashCode()` method would return the same value for both of them.

This is definitely a job for `IdentityHashMap`, which would be able to distinguish between the two nodes with the same content string. Let's update our code to use `IdentityHashMap` instead of `HashMap`:

```
public void dump() {
  dump( "", new IdentityHashMap() );
}
```

Because `IdentityHashMap` treats these two nodes as separate nodes, it actually dumps the last node out instead of mistaking it for the other node with the same content string. Here's the output from the updated program:

```
Node: a [7704795/97]
  Node: b [1375836/98]
    (no children)
  Node: c [4687246/99]
    Node: d [2866566/100]
      (no children)
    Node: a [7704795/97]
      (loop)
    Node: a [7704795/97]
      Node: b [1375836/98]
        (no children)
      Node: e [15805518/101]
        (no children)
```

This is one of the rare times when you need an `IdentityHashMap` instead of a regular `HashMap`. Generally speaking, you'll use an `IdentityHashMap` when you are dealing with objects as abstract `Objects` rather than as the things they are intended to be. In this example, the `Node` objects could be used for just about anything—parsing the code of a programming language, representing computers in a network, and so on. In that capacity, two `Node` objects containing the string "a" might be the same. But as pure `Objects` they are definitely different objects, and our `dump()` routine must treat them as such.

Before the arrival of JDK 1.4 and `IdentityHashMap`, you could have solved this by changing your definition of `Node.equals()` to distinguish between nodes with the same content string. (You'll note that the constructor `Node` actually takes a content `Object` as its argument, rather than requiring that it be a `String`. This object can be anything you wish to put inside a graph, and it can have any implementation of `equals()` and `hashCode()` that you want.)

However, you might have good reasons for using your particular implementation of `equals()`, and you may not want to change it. In this case, `System.identityHashCode()` can be used to distinguish between two separate objects that have the same hash code, and `==` can be used instead of `equals()`. In fact, we're doing this very thing, indirectly, when we use `IdentityHashMap`, because `IdentityHashMap` uses `System.identifyHashCode()` to distinguish between objects.

8.4 The RandomAccess interface

The `RandomAccess` interface is a *marker interface*, which means that it exists only to mark a class as having a certain property. As such, it is an *empty* interface—it has no

methods. A class implements such an interface merely to advertise that it has a certain property. In this case, a `List` implementing `RandomAccess` advertises that the list supports efficient random access.

Here, random access means the use of `get()` and `set()` methods, in contrast to using the list's `Iterator`. In general, an `Iterator` is supposed to iterate through a list using the fastest possible method, while the `get()` and `set()` methods may or may not be efficient. If they are efficient, then the list should implement `RandomAccess`.

Code that does list processing and that would benefit from the list supporting fast random access should check the list first to see if it does in fact implement the `RandomAccess` interface:

```
if (!(list instanceof RandomAccess)) {
}
```

If the list does *not* implement `RandomAccess`, the code may have to use an alternative algorithm to process the list.

Here's an example of this in action. In listing 8.5, the `ListTransform` interface specifies a generic list-transforming routine that rearranges the elements within a list. `RandomTransform` is an example class that implements `ListTransform`; what it does is take a list and randomly swap its elements in order to randomize it (much like the `shuffle()` method in the `Collections` class). This is just a simple example transform—all that matters is that the transform require random access to be efficient.

The `RandomAccessifier` class simply applies a `ListTransform` to a `List`—but with an added bit of cleverness. If the `List` doesn't implement `RandomAccess`, it is first copied into a `List` that does. The transform is then applied, and the data is then copied back into the original list. If the `List` does implement `RandomAccess`, then the list is transformed in-place.

Listing 8.5 RandomAccessifier.java

(see \Chapter8\RandomAccessifier.java)

```
import java.util.*;

public class RandomAccessifier
{
  interface ListTransform
  {
    public void transform( List list );
  }

  static class RandomTransform implements ListTransform
  {
    private int count;
    private Random rand = new Random();
```

Interface for a list-transformer

ListTransform randomly rearranges elements

```
    public RandomTransform( int count ) {
      this.count = count;
    }

    public void transform( List list ) {
      for (int i=0; i<count; ++i) {
        int ai = rand.nextInt( list.size() );
        int bi = rand.nextInt( list.size() );
        Collections.swap( list, ai, bi );
      }
    }
  }

  public static void transform( List list,
                                ListTransform transform ) {
    List origList = list;
    boolean ra = (list instanceof RandomAccess);
    if (!ra) {
      System.out.println( "Converting to RA" );
      list = new ArrayList( origList.size() );
      for (Iterator it=origList.iterator(); it.hasNext();) {
        list.add( it.next() );
      }
    }
    transform.transform( list );
    if (!ra) {
     origList.clear();
      int size = list.size();
      for (Iterator it=list.iterator(); it.hasNext();) {
        origList.add( it.next() );
      }
    }
  }

  static public void main( String args[] ) {
    List list = new LinkedList();
    for (int i=0; i<100; ++i) {
      list.add( new Integer( i ) );
    }

    RandomTransform rt = new RandomTransform( 10000000 );
    transform( list, rt );

    for (Iterator it=list.iterator(); it.hasNext();) {
      System.out.print( it.next()+" " );
    }
    System.out.println( "" );
  }
}
```

Move the elements into a RandomAccess list (1)

Copy into an ArrayList (2)

(3) **Move the elements back into the original list**

1 3 Note that we avoid using the `get()` and `set()` methods when copying out of the original array, and when copying back into it. This is for the very same reason we are doing the copy in the first place: these methods aren't very efficient.

Instead, we use an `Iterator` to copy the elements out of the array. To copy the elements back in, we actually empty the array with the `clear()` method, and then `add()` the elements back in one-by-one.

2 `ArrayList` is one of the two classes in JDK 1.4 that implements the `RandomAccess` interface, the other being `Vector`. It can safely be assumed that future releases of the JDK will take care to follow the convention of implementing the `RandomAccess` interface in classes for which it is appropriate.

The `main()` routine in the preceding example applies the `RandomTransform` to a `LinkedList`, which does *not* implement the `RandomAccess` interface. Using the `RandomAccessifier` to speed things up results in a significant speed improvement.

8.5 Summary

There's not a whole lot going on in the Collections Framework in this release, mostly because the Collections Framework is already so powerful. The new features added in this release could probably be written by applications programmers themselves, but having the implementations built into the core release standardizes their properties and allows for extra optimizations.

The addition of the `RandomAccess` marker interfaces heralds the addition of similar markers in the future—markers such as these will permit customization and specialization of standard collection operations that will allow for incremental increases in efficiency even as the core contracts of the Collections Framework classes are maintained.