CHAPTER

15 Inner Classes and Implementation of ArrayList

THIS chapter begins our discussion of the implementation of standard data structures. One of the simplest data structures is the `ArrayList` that is part of the Collections API. In Part I (specifically Figure 3.9 and Figure 4.23) we have already seen skeletons of the implementation, so in this chapter we concentrate on the details of implementing the complete class, with the associated iterators. In doing so, we make use of an interesting Java syntactic creation, the *inner class*. We discuss the inner class in this chapter, rather than in Part I (where other syntactic elements are introduced) because we view the inner class as a Java implementation technique, rather than a core language feature.

In this chapter, we will see:

- The uses and syntax of the inner class
- An implementation of a new class called the `AbstractCollection`
- An implementation of the `ArrayList` class.

15.1 Iterators and Nested Classes

We begin by reviewing the simple iterator implementation first described in Section 6.2. Recall that we defined a simple iterator interface, that mimics the standard Collections API `Iterator`, and this interface is shown in Figure 15.1.

```
1 package weiss.ds;
2
3 public interface Iterator
4 {
5     boolean hasNext( );
6     Object next( );
7 }
```

Figure 15.1 The `Iterator` interface from Section 6.2

```
1 // An iterator class that steps through a MyContainer.
2
3 package weiss.ds;
4
5 class MyContainerIterator implements Iterator
6 {
7     private int current = 0;
8     private MyContainer container;
9
10    MyContainerIterator( MyContainer c )
11      { container = c; }
12
13    public boolean hasNext( )
14      { return current < container.size; }
15
16    public Object next( )
17      { return container.items[ current++ ]; }
18 }
```

Figure 15.2 Implementation of the `MyContainerIterator`, from Section 6.2

```
 1 package weiss.ds;
 2
 3 public class MyContainer
 4 {
 5     Object [ ] items;
 6     int size;
 7
 8     public Iterator iterator( )
 9       { return new MyContainerIterator( this ); }
10
11     // Other methods not shown.
12 }
```

Figure 15.3 The `MyContainer` class from Section 6.2

We then define two classes: the container and its iterator. Each container class is responsible for providing an implementation of the iterator interface. In our case, the implementation of the iterator interface is provided by the `MyContainerIterator` class, shown in Figure 15.2. The `MyContainer` class, shown in Figure 15.3, provides a factory method that creates an instance of `MyContainerIterator`, and returns this instance using the interface type `Iterator`. Figure 15.4 provides a main that illustrates the use of the container/iterator combination. Figures 15.1 to 15.4 simply replicate Figures 6.5 to 6.8 in the original iterator discussion from Section 6.2.

```
1      public static void main( String [ ] args )
2      {
3          MyContainer v = new MyContainer( );
4
5          v.add( "3" );
6          v.add( "2" );
7
8          System.out.println( "Container contents: " );
9          Iterator itr = v.iterator( );
10         while( itr.hasNext( ) )
11             System.out.println( itr.next( ) );
12     }
```

Figure 15.4 `main` method to illustrate iterator design from Section 6.2

```
1  package weiss.ds;
2
3  public class MyContainer
4  {
5      private Object [ ] items;
6      private int size = 0;
7      // Other methods for MyContainer not shown
8
9      public Iterator iterator( )
10       { return new LocalIterator( this ); }
11
12     // The iterator class as a nested class
13     private static class LocalIterator implements Iterator
14     {
15         private int current = 0;
16         private MyContainer container;
17
18         LocalIterator( MyContainer c )
19           { container = c; }
20
21         public boolean hasNext( )
22           { return current < container.size; }
23
24         public Object next( )
25           { return container.items[ current++ ]; }
26     }
27 }
```

Figure 15.5 Iterator design using nested class.

This design hides the iterator class implementation, because `MyContainerIterator` is not a public class. Thus the user is forced to pro-

gram to the `Iterator` interface, and does not have access to the details of how the iterator was implemented—the user cannot even declare objects of type `weiss.ds.MyContainerIterator`. However, it still exposes more details than we usually like. In the `MyContainer` class, the data is not private, and the corresponding iterator class, while not public, is still package visible. We can solve both problems by using nested classes: we simply move the iterator class inside of the container class. At that point the iterator class is a member of the container class, and thus it can be declared as a private class and its methods can access private data from `MyContainer`. The revised code is illustrated in Figure 15.5; with only a stylistic change of renaming `MyContainerIterator` as `LocalIterator`. No other changes are required.

15.2 Iterators and Inner Classes

An *inner class* is similar to a nested class in that it is a class inside another class, and is declared using the same syntax as a nested class, except that it is not a `static` class. An inner class always contains an implicit reference to the outer object that created it.

In the Section 15.1, we used a nested class to further hide details. In addition to nested classes, Java provides inner classes. An *inner class* is similar to a nested class in that it is a class inside another class, and is treated as a member of the outer class for visibility purposes. An inner class is declared using the same syntax as a nested class, except that it is not a static class. In other words, the `static` qualifier is missing in the inner class declaration.

Before getting into the inner class specifics, let us look at the problem that they are designed to solve. Figure 15.6 illustrates the relationship between the iterator and container classes that were written in the previous section. Each instance of the `LocalIterator` maintains a reference to the container over which it is iterating and a notion of the iterator's current position. The relationship that we have is that each `LocalIterator` must be associated with exactly one instance of `MyContainer`. It is impossible for the `container` reference in any iterator to be `null`, and the iterator's existence makes no sense without knowing which `MyContainer` object caused its creation.

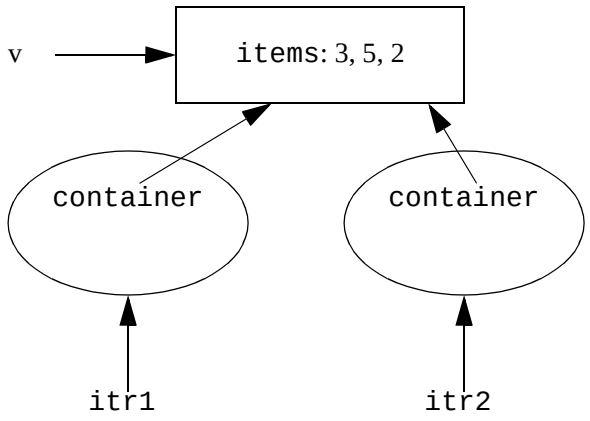

Figure 15.6 Iterator/container relationship

Since we know that `itr1` must be tied to one and only one iterator, it seems that the expression `container.items` is redundant: if the iterator could only remember the container that constructed it, we wouldn't have to keep track of it ourselves. And if it remembered it, we might expect that if inside of the `LocalIterator` we referred to `items`, then since the `LocalIterator` does not have an `items` field, the compiler (and runtime system) would be smart enough to deduce that we are talking about the `items` field of the `MyContainer` object that caused the construction of this particular `LocalIterator`. This is exactly what an inner class does, and what distinguishes it from a nested class.

The big difference between an inner class and a nested class is that when an instance of an inner class object is constructed, there is an implicit reference to the outer class object that caused its construction. This implies that an inner class

object cannot exist without an outer class object for it to be attached to, with an exception being if it is declared in a static method (because local and anonymous classes are technically inner classes), a detail we will discuss later.

If the name of the outer class is `Outer`, then the implicit reference is `Outer.this`.

If the name of the outer class is `Outer`, then the implicit reference is `Outer.this`. Thus, if `LocalIterator` was declared as an instance inner class (i.e. the `static` keyword was removed), then the `MyContainer.this` reference could be used to replace the `container` reference that the iterator is storing. The picture in Figure 15.7 illustrates that the structure would be identical. A revised class is shown in Figure 15.8.

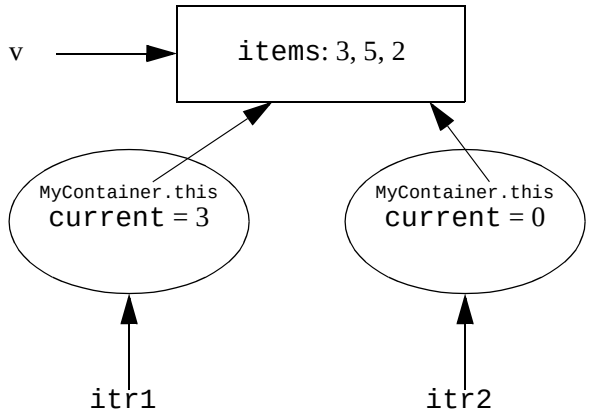

Figure 15.7 Iterator/container with inner classes

```
 1 package weiss.ds;
 2
 3 public class MyContainer
 4 {
 5     private Object [ ] items;
 6     private int size = 0;
 7
 8     // Other methods for MyContainer not shown
 9
10     public Iterator iterator( )
11       { return new LocalIterator( ); }
12
13     // The iterator class as an inner class
14     private class LocalIterator implements Iterator
15     {
16         private int current = 0;
17
18         public boolean hasNext( )
19           { return current < MyContainer.this.size; }
20
21         public Object next( )
22           { return MyContainer.this.items[ current++ ]; }
23     }
24 }
```

Figure 15.8 Iterator design using inner class.

In the revised implementation, observe that `LocalIterator` no longer has an explicit reference to a `MyContainer`, and also observe that its constructor is no longer necessary, since it only initialized the `MyContainer` reference. Finally, Figure 15.9 illustrates that just as using `this` is optional in an instance method, the `Outer.this` reference is also optional if there is no name clash. Thus, `MyContainer.this.size` can be shortened to `size`, as long as there is no other variable named size that is in a closer scope.

```
13      // The iterator class as an inner class
14      private class LocalIterator implements Iterator
15      {
16          private int current = 0;
17
18          public boolean hasNext( )
19            { return current < size; }
20
21          public Object next( )
22            { return items[ current++ ]; }
23      }
```

Figure 15.9 Inner class; `Outer.this` may be optional

Local classes and anonymous classes do not specify whether they are `static`, and they are always technically considered inner classes. However, if such a class is declared in a static method, it has no implicit outer reference (and thus behaves like a nested class), whereas if it is declared inside an instance method, its implicit outer reference is the invoker of the method.

The addition of inner classes requires a significant set of rules, many of which attempt to deal with language corner cases and dubious coding practices. For instance, suppose we suspend belief for a minute and imagine that `LocalIterator` is public. We do so only to illustrate the complications that the language designers face when adding a new language feature. Under this assumption the iterator's type is `MyContainer.LocalIterator`, and since it is visible, one might expect that

```
MyContainer.LocalIterator itr =
                    new MyContainer.LocalIterator( );
```

is legal, since like all classes, it has a default a zero-parameter constructor. However, this cannot possibly work, since there is no way to initialize the implicit ref-

erence. Which `MyContainer` is `itr` referring to? We need some syntax, that won't conflict with any other language rules. Here's the rule: If there is a container `c`, then `itr` could be constructed using a bizarre syntax invented for just this case, in which the outer object in effect invokes `new`:

```
MyContainer.LocalIterator itr = c.new LocalIterator( );
```

Notice that this implies that in an instance factory method, `this.new` is legal, and shorthands to the more conventional `new` seen in a factory method. If you find yourself using the bizarre syntax, then most likely you have a bad design. In our example, once `LocalIterator` is private, this entire issue goes away, and if `LocalIterator` is not private, there is little reason to use an inner class in the first place.

There are also other rules, some of which are arbitrary. Private members of the inner or nested class are public to the outer class. To access any member of an inner class, the outer class only needs to provide a reference to an inner class instance and use the dot operator, as is normal for other classes. Thus inner and nested classes are considered part of the outer class.

Both inner and nested classes can be final, or they can be abstract, or they can be interfaces (but interfaces are always static, because they cannot have any data, including an implicit reference), or they can be none of these. Inner classes may not have static fields or methods, except for static final fields. Interfaces may have nested classes or interfaces. Finally, when you compile the above example, you will see that the compiler generates a class file named `MyCon-`

`tainer$LocalIterator.class`, which would have to be included in any distribution to clients. In other words, each inner and nested class is a class, and has a corresponding class file. Anonymous classes use numbers instead of names.

```
1  package weiss.util;
2
3  /**
4   * AbstractCollection provides default implementations for
5   * some of the easy methods in the Collection interface.
6   */
7  public abstract class AbstractCollection implements Collection
8  {
9      /**
10      * Tests if this collection is empty.
11      * @return true if the size of this collection is zero.
12      */
13     public boolean isEmpty( )
14     {
15         return size( ) == 0;
16     }
17
18     /**
19      * Change the size of this collection to zero.
20      */
21     public void clear( )
22     {
23         Iterator itr = iterator( );
24         while( itr.hasNext( ) )
25         {
26             itr.next( );
27             itr.remove( );
28         }
29     }
30
31     /**
32      * Obtains a primitive array view of the collection.
33      * @returns the primitive array view.
34      */
35     public Object [] toArray( )
36     {
37         Object [] copy = new Object[ size( ) ];
38
39         Iterator itr = iterator( );
40         int i = 0;
41
42         while( itr.hasNext( ) )
43             copy[ i++ ] = itr.next( );
44
45         return copy;
46     }
```

Figure 15.10 Sample implementation of `AbstractCollection` (Part 1)

```
47      /**
48       * Returns true if this collection contains x.
49       * If x is null, returns false.
50       * (This behavior may not always be appropriate.)
51       * @param x the item to search for.
52       * @returns true if x is not null and is found in
53       * this collection.
54       */
55      public boolean contains( Object x )
56      {
57          if( x == null )
58              return false;
59
60          Iterator itr = iterator( );
61          while( itr.hasNext( ) )
62              if( x.equals( itr.next( ) ) )
63                  return true;
64
65          return false;
66      }
67
68      /**
69       * Removes non-null x from this collection.
70       * @param x the item to remove.
71       * @returns true if remove succeeds.
72       */
73      public boolean remove( Object x )
74      {
75          if( x == null )
76              return false;
77
78          Iterator itr = iterator( );
79          while( itr.hasNext( ) )
80              if( x.equals( itr.next( ) ) )
81              {
82                  itr.remove( );
83                  return true;
84              }
85
86          return false;
87      }
88  }
```

Figure 15.11 Sample implementation of `AbstractCollection` (Part 2)

15.3 The AbstractCollection Class

The AbstractCollection implements some of the methods in the Collection interface.

Before we implement the `ArrayList` class, observe that some of the methods in the `Collection` interface can be easily implemented in terms of others. For instance, `isEmpty` is easily implemented by checking if the size is 0. Rather than doing so in `ArrayList`, `LinkedList`, and all the other concrete implementations, it would be preferable to do this once, and use inheritance to obtain `isEmpty`. We could even override `isEmpy` if it turns out that for some collections there is a faster way of performing `isEmpty` than computing the current size. However, we cannot implement `isEmpty` in the `Collection` interface; this can only be done in an abstract class. This will be the `AbstractCollection` class. To simplify implementations, programmers designing new collections classes can extend the `AbstractCollection` class rather than implementing the `Collection` interface A sample implementation of `AbstractCollection` is shown in Figures 15.10 and 15.11.

The Collections API also defines additional classes such as `AbstractList`, `AbstractSequentialList`, and `AbstractSet`.We have chosen not to implement those, in keeping with our intention of providing a simplified subset of the Collections API. If, for some reason you are implementing your own collections and extending the Java 1.2 Collections API, you should extend the most specific abstract class.

15.4 Implementation of `ArrayList` with an Iterator

The various `ArrayList` classes shown in Part I were not iterator-aware. This section provides an implementation of `ArrayList` that we will place in `weiss.util`, and includes support for bidirectional iterators. In order to keep the amount of code somewhat manageable, we have stripped out the bulk of the Javadoc comments. They can be found in the online code.

The implementation is found in Figures 15.12 to 15.15. At line 3 we see that `ArrayList` extends the `AbstractCollection` abstract class, and at line 4 `ArrayList` declares that it implements the `List` interface.

The internal array, `theItems`, and collection size, `theSize`, are declared at lines 9 and 10, respectively. More interesting is `modCount`, which is declared at line 11. `modCount` represents the number of structural modifications (`add`s, `remove`s) made to the `ArrayList`. The idea is that when an iterator is constructed, the iterator saves this value in its data member `expectedModCount`. When any iterator operation is performed, the iterator's `expectedModCount` member is compared with the `ArrayList`'s `modCount`, and if they disagree, a `ConcurrentModificationException` can be thrown.

Line 16 illustrates the typical constructor that performs a shallow copy of the members in another collection, simply by stepping through the collection and calling `add`. The `clear` method, started at line 27, initializes the `ArrayList` and can be called from the constructor. It also resets `theItems`, which allows the garbage collector to reclaim all the otherwise unreferenced objects that were

in the ArrayList. The remaining routines in Figure 15.12 are relatively straightforward.

Figure 15.13 implements the remaining methods that do not depend on iterators. findPos is a private helper that returns the position of an object that is either being removed or subjected to a contains call. Extra code is present because it is legal to add null to the ArrayList, and if we were not careful, the call to equals at line 60 could have generated a NullPointerException. Observe that both add and remove will result in a change to modCount.

In Figure 15.14 we see the two factory methods that return iterators, and we see the beginning of the implementation of the ListIterator interface. Observe that ArrayListIterator IS-A ListIterator and ListIterator IS-A Iterator. So ArrayListIterator can be returned at lines 103 and 106.

In the implementation of ArrayListIterator, done as a private inner class, we maintain the current position at line 111. The current position represents the index of element that would be returned by calling next. At line 112 we declare the expectedModCount member. Like all class members, it is initialized when an instance of the iterator is created (immediately prior to calling the constructor); modCount is a shorthand for ArrayList.this.modCount. The two Boolean instance members that follow are flags used to verify that a call to remove is legal.

```
 1 package weiss.util;
 2
 3 public class ArrayList extends AbstractCollection
 4                         implements List
 5 {
 6     private static final int DEFAULT_CAPACITY = 10;
 7     private static final int NOT_FOUND = -1;
 8
 9     private Object [] theItems;
10     private int theSize;
11     private int modCount = 0;
12
13     public ArrayList( )
14       { clear( ); }
15
16     public ArrayList( Collection other )
17     {
18         clear( );
19         Iterator itr = other.iterator( );
20         while( itr.hasNext( ) )
21             add( itr.next( ) );
22     }
23
24     public int size( )
25       { return theSize; }
26
27     public void clear( )
28     {
29         theSize = 0;
30         theItems = new Object[ DEFAULT_CAPACITY ];
31         modCount++;
32     }
33
34     public Object get( int idx )
35     {
36         if( idx < 0 || idx >= size( ) )
37             throw new ArrayIndexOutOfBoundsException( );
38         return theItems[ idx ];
39     }
40
41     public Object set( int idx, Object newVal )
42     {
43         if( idx < 0 || idx >= size( ) )
44             throw new ArrayIndexOutOfBoundsException( );
45         Object old = theItems[ idx ];
46         theItems[ idx ] = newVal;
47         return old;
48     }
49
50     public boolean contains( Object x )
51       { return findPos( x ) != NOT_FOUND; }
```

Figure 15.12 `ArrayList` implementation (part 1)

```
52      private int findPos( Object x )
53      {
54          for( int i = 0; i < size( ); i++ )
55              if( x == null )
56              {
57                  if( theItems[ i ] == null )
58                      return i;
59              }
60              else if( x.equals( theItems[ i ] ) )
61                  return i;
62
63          return NOT_FOUND;
64      }
65
66      public boolean add( Object x )
67      {
68          if( theItems.length == size( ) )
69          {
70              Object [] old = theItems;
71              theItems = new Object[ theItems.length * 2 + 1 ];
72              for( int i = 0; i < size( ); i++ )
73                  theItems[ i ] = old[ i ];
74          }
75          theItems[ theSize++ ] = x;
76          modCount++;
77          return true;
78      }
79
80      public boolean remove( Object x )
81      {
82          int pos = findPos( x );
83
84          if( pos == NOT_FOUND )
85              return false;
86          else
87          {
88              remove( pos );
89              return true;
90          }
91      }
92
93      public Object remove( int idx )
94      {
95          Object removedItem = theItems[ idx ];
96          for( int i = idx; i < size( ) - 1; i++ )
97              theItems[ i ] = theItems[ i + 1 ];
98          theSize--;
99          modCount++;
100         return removedItem;
101     }
```

Figure 15.13 `ArrayList` implementation (part 2)

```
102     public Iterator iterator( )
103       { return new ArrayListIterator( 0 ); }
104
105     public ListIterator listIterator( int idx )
106       { return new ArrayListIterator( idx ); }
107
108     // This is the implementation of the ArrayListIterator.
109     private class ArrayListIterator implements ListIterator
110     {
111         private int current;
112         private int expectedModCount = modCount;
113         private boolean nextCompleted = false;
114         private boolean prevCompleted = false;
115
116         ArrayListIterator( int pos )
117         {
118             if( pos < 0 || pos > size( ) )
119                 throw new IndexOutOfBoundsException( );
120             current = pos;
121         }
122
123         public boolean hasNext( )
124         {
125             if( expectedModCount != modCount )
126                 throw new ConcurrentModificationException( );
127             return current < size( );
128         }
129
130         public boolean hasPrevious( )
131         {
132             if( expectedModCount != modCount )
133                 throw new ConcurrentModificationException( );
134             return current > 0;
135         }
```

Figure 15.14 `ArrayList` implementation (part 3)

The `ArrayListIterator` constructor is declared package visible; thus it is usable by the `ArrayList`. Of course it could be declared public, but there is no reason to do so. Both `hasNext` and `hasPrevious` verify that there have been no external structural modifications since the iterator was created, throwing an exception if the `ArrayList` `modCount` does not match the `ArrayListIterator` `expectedModCount`.

The ArrayListIterator class is completed in Figure 15.15. next and previous are mirror image symmetries. Examining next, we see first a test at line 138 to make sure we have not exhausted the iteration (implicitly this tests for structural modifications also). We then set nextCompleted to true to allow remove to succeed, and then we return the array item that current is examining, advancing current after its value has been used.

previous is similar, except that we must lower current's value first. This is because when traversing in reverse, if current equals the container size, we have not yet started the iteration, and when current equals zero, we have completed the iteration (but can remove the item in position, if the prior operation was previous). Observe that next followed by previous yields identical items.

Finally, we come to remove, which is extremely tricky because the semantics of remove depend on which direction the traversal is proceeding. In fact, this probably suggests a bad design in the Collections API: method semantics should not depend so strongly on which methods have been called prior to it. But remove is what it is, so we have to implement it.

```
136         public Object next( )
137         {
138             if( !hasNext( ) )
139                 throw new NoSuchElementException( );
140             nextCompleted = true;
141             prevCompleted = false;
142             return theItems[ current++ ];
143         }
144
145         public Object previous( )
146         {
147             if( !hasPrevious( ) )
148                 throw new NoSuchElementException( );
149             prevCompleted = true;
150             nextCompleted = false;
151             return theItems[ --current ];
152         }
153
154         public void remove( )
155         {
156             if( expectedModCount != modCount )
157                 throw new ConcurrentModificationException( );
158
159             if( nextCompleted )
160                 ArrayList.this.remove( --current );
161             else if( prevCompleted )
162                 ArrayList.this.remove( current );
163             else
164                 throw new IllegalStateException( );
165
166             prevCompleted = nextCompleted = false;
167             expectedModCount++;
168         }
169     }
170 }
```

Figure 15.15 `ArrayList` implementation (part 4)

The implementation of `remove` begins with the test for structural modification at line 156. If the prior iterator state change operation was a `next`, as evidenced by the test at line 159 showing that `nextCompleted` is true, then we call the `ArrayList` `remove` method (started at line 93 in Figure 15.13) that takes an index as a parameter. The use of `ArrayList.this.remove` is required because the local version of `remove` hides the outer class version.

Because we have already advanced pass the item to be removed, we must remove the item in position `current-1`. This slides the next item from `current` to `current-1` (since the old `current-1` position has now been removed) so we use the expression `--current` in line 160.

When traversing the other direction, we are sitting on the last item that was returned, so we simply pass `current` as a parameter to the outer `remove`. After it returns, the elements in higher indices are slid one index lower, so `current` is sitting on the correct element, and can be used in the expression at line 162.

In either case, we cannot do another `remove` until we do a `next` or `previous`, so at line 166 we clear both flags. Finally, at line 167, we increase the value of `expectedModCount` to match the container's. Observe that this is increased only for this iterator, so any other iterators are now invalidated.

This class, which is perhaps the simplest of the Collections API classes that contains iterators illustrates why in Part IV we elect to begin with a simple protocol, and then provide more complete implementations at the end of the chapter.

Summary

This chapter introduced the inner class, which is a Java technique that is commonly used to implement iterator classes. Each instance of an inner class corresponds to exactly one instance of an outer class and automatically maintains a reference to the outer class object that caused its construction. A nested class relates two types to each other, while an inner class relates two objects to each other. The inner class is used in this chapter to implement the `ArrayList`.

The next chapter illustrates implementations of stacks and queues.

Objects of the Game

`AbstractCollection` Implements some of the methods in the `Collection` interface. (619)

inner class A class inside a class, that is useful for implementing the iterator pattern. The inner class always contains an implicit reference to the outer object that created it. (610)

Common Error

1. An instance inner class cannot be constructed without an outer object. This is most easily done with a factory method in the outer class. It is common to forget the word `static` when declaring a nested class, and this will often generate a difficult to understand error related to this rule.

On the Internet

The following files are available:

MyContainerTest.java The test program for the final iterator example that uses inner classes, as shown in Section 15.2. **Iterator.java** and **MyContainer.java** are both found in the `weiss.ds` package online.

AbstractCollection.java Contains the code in Figures 15.10 and 15.11.

ArrayList.java Contains the code in Figures 15.12 to 15.15.

Exercises

In Short

15.1. What is the difference between a nested class and an inner class?

15.2. Are private members of an inner (or nested) class visible to methods in the outer class?

15.3. In Figure 15.16, are the declarations of `a` and `b` legal? Why or why not?.

15.4. In Figure 15.16, (assuming illegal code is fixed) how are objects of type `Inner1` and `Inner2` created (you may suggest additional members)?

```
1  class Outer
2  {
3      private int x = 0;
4      private static int y = 37;
5
6      private class Inner1 implements SomeInterface
7      {
8          private int a = x + y;
9      }
10     private static class Inner2 implements SomeInterface
11     {
12         private int b = x + y;
13     }
14 }
```

Figure 15.16 Code for Figure 15.3

In Theory

15.5. Suppose an inner class *I* is declared public in its outer class *O*. Why might unusual syntax be required to declare a class *E* that extends *I* but is

declared as a top-level class? (The required syntax is even more bizarre than what was seen for `new`, but often requires bad design to be needed.)

15.6. What is the running time of `clear`, as implemented for `ArrayList`? What would be the running time if the inherited version from `AbstractCollection` was used instead?

In Practice

15.7. Add both the `previous` and `hasPrevious` methods to the final version of the `MyContainer` class.

15.8. Assume that we would like an iterator that implements the `isValid`, `advance`, and `retrieve` set of methods, but all we have is the standard `java.util.Iterator` interface.

a. What pattern describes the problem we are trying to solve?

b. Design a `BetterIterator` class, and then implement it in terms of `java.util.Iterator`.

15.9. Figure 15.17 contains two proposed implementations of `clear` for `AbstractCollection`. Do either work?

15.10. Provide an implementation of `toString` for `AbstractCollection`. The running time should be linear in the size of the collection.

Programming Projects

15.11. The `Collection` interface in the Java Collections API defines methods `removeAll`, `addAll`, and `containsAll`. Add these methods to the

`Collection` interface and provide implementations in `AbstractCollection`.

```
1     public void clear( )   // Version #1
2     {
3         Iterator itr = iterator( );
4         while( !isEmpty( ) )
5             remove( itr.next( ) );
6     }
7
8     public void clear( )   // Version #2
9     {
10        while( !isEmpty( ) )
11            remove( iterator( ).next( ) );
12    }
```

Figure 15.17 Proposed implementations of `clear` for `AbstractCollection`

15.12. `Collections.unmodifiableCollection` takes a `Collection` and returns an immutable `Collection`. Implement this method. To do so, you will need to use a class inside a method. The class implements the `Collection` interface, and throws an `UnsupportedOperationException` for all mutating methods. For other methods, it forwards the request to the `Collection` being wrapped. You will also have to hide an unmodifiable iterator.

15.13. Two `Collection` objects are equal if either both implement the `List` interface and contain the same items in the same order or both implement the `Set` interface and contain the same items in any order. Otherwise, the `Collection` objects are not equal. Provide, in `AbstractCollection`, an implementation of `equals` that follows

this general contract. Additionally, provide a `hashCode` method in `AbstractCollection` that follows the general contract of `hashCode` (do this by using an iterator and adding the `hashCode`s of all the entries. Watch out for `null` entries.)