5.3 GENERIC CLASSES

Writing a Generic Class

A generic *method* is a method that depends on an unspecified underlying data type. In a similar way, when an entire *class* depends on an underlying data type, the class can be implemented as a **generic class**, resulting in the same typechecking advantages that you have seen for generic methods. For example, in this section, we'll convert our `IntArrayBag` (from Chapter 3) to a generic class, which means that the data type of the objects in the bag is left unspecified.

A generic class is written by putting a generic type parameter in angle brackets immediately after the class name in the class's implementation. For example, the implementation of a generic bag class might begin like this:

```
public class ArrayBag<E> implements Cloneable...
```

The name `ArrayBag` indicates that the bag will store its elements in an array. We can choose the name of the generic type parameter to be anything we like, but the designers of Java suggest using a single capital letter, such as `E`, to indicate that it is the unknown class of an "element" in the bag. Throughout the rest of the implementation of the bag, we can use the unknown class, `E`, as if it were any other class name. For example, our `ArrayBag` will have two private member variables, one of which is an array to hold the elements of type `E`, as shown here:

```
public class ArrayBag<E> implements Cloneable
{
   private E[ ] data;      // An array to hold the bag's elements
   private int manyItems; // Number of items in the bag
```

We'll see the entire implementation shortly, and it won't differ much from the original `IntArrayBag`, but first let's look at how a program uses a generic class.

Using a Generic Class

A program that wants to use a generic class must explicitly specify what class will be used for the generic type parameter. This process is called **instantiating** the generic type parameter. The syntax uses the name of the generic class followed by the name of the class that you want to use for the generic type parameter. For example, these lines will create a bag of strings and a bag of integers (using the default constructor in both cases):

```
ArrayBag<String> sbag = new ArrayBag<String>( );
ArrayBag<Integer> ibag = new ArrayBag<Integer>( );
```

We can then activate member functions such as `sbag.add("Thunder")` or `ibag.add(42)`. This `ibag` activation will autobox the 42, so that the argument is an `Integer` object rather than a primitive `int`. However, the compiler will

generate a typechecking error for a statement such as `ibag.add("Thunder")`. You cannot add a `String` object to a bag of `Integer` objects.rs.

PITFALL

GENERIC CLASS RESTRICTIONS

In a generic class, the type used to instantiate the generic type parameter must be a class (not a primitive type). In addition, within the implementation of the generic class, you may not call a constructor for the generic type, nor may you create a new array of elements of that type.

These restrictions are identical to the restrictions for a generic method (page 251), and are a consequence of the compilation method that Java uses for generic classes.

Details for Implementing a Generic Class

Figure 5.1 on page 260 gives the complete implementation of the generic `ArrayBag` class. The implementation is mostly unsurprising: We took the original `IntArrayBag` implementation, changed the start to `ArrayBag<E>`, and throughout the implementation we use the unknown type `E` as the type of element in the bag (rather than `int`). Still, there are a few issues that you need to examine, and we'll discuss those issues next.

Creating an Array to Hold Elements of the Unknown Type

Within the generic class implementation, we are not allowed to create a new array of objects of the unknown type `E`. This causes a problem for our bag because we need an array that holds references to objects of type `E`. The solution is to create an array of Java objects, and typecast that array to an array of type `E`, as shown here in the typecast `(E[])` from one of the `ArrayBag` constructors:

```
public ArrayBag( )
{
   final int INITIAL_CAPACITY = 10;
   manyItems = 0;L
   data = (E[]) new Object[INITIAL_CAPACITY];
}
```

Some compilers will produce a warning for a typecast such as this because the compiler cannot guarantee at compile time that the data array will always contain `E` objects, rather than some other kind of object. So, it will be up to us in our programming of the `ArrayBag` to ensure that we don't misuse the data array.

Using ArrayBag as the Type of a Parameter or Return Value

The bag has several methods which have bags as parameters or as the return type. For example, our `IntArrayBag` has an `addAll` method:

```
public void addAll(IntArrayBag addend)...
```

When a program activates `b1.addall(b2)`, all the integers from `b2` are put in the bag `b1`.

For the generic bag, the way to implement such a parameter is to specify its data type as `ArrayBag<E>` (rather than just `ArrayBag`):

```
public void addAll(ArrayBag<E> addend)...
```

This will allow us to activate `b1.addAll(b2)` for two bags, `b1` and `b2`, but only if the type of elements in `b1` is the same as the type of elements in `b2`.

For our array bag, all the uses of the `ArrayBag` data type within its own implementation will be written as `ArrayBag<E>`.

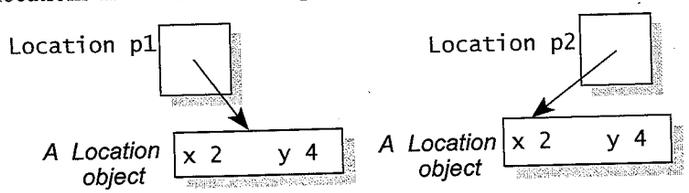
Counting the Occurrences of an Object

Counting the occurrences of an object in a bag requires some thought. Here's an example to get you thinking. Suppose you construct a bag of `Location` objects and create two identical `Location` objects with coordinates of x=2, y=4 (using the `Location` class from Section 2.4). The locations are then added to the bag:

```
ArrayBag<Location> spots = new ArrayBag<Location>( );
Location p1 = new Location(2, 4);  // x=2 and y=4
Location p2 = new Location(2, 4);  // Also at x=2 and y=4

spots.add(p1);
spots.add(p2);
```

The two locations are identical but separate objects, as shown in this drawing:



Keep in mind that the boolean expression `(p1 == p2)` is `false` because "==" returns `true` only if `p1` and `p2` refer to the exact same object (as opposed to two separate objects that happen to contain identical values). On the other hand, the `Location` class has an `equals` method with this specification:

♦ **equals (from the Location class)**

`public boolean equals(Object obj)`

Compare this `Location` to another object for equality.

Parameters:

`obj` – an object with which this `Location` is compared

Returns:

A return value of `true` indicates that `obj` refers to a `Location` object with the same value as this `Location`. Otherwise, the return value is `false`.

Both `p1.equals(p2)` and `p2.equals(p1)` are `true`.

So here's the question to get you thinking: With both locations in the bag, what is the value of `spots.countOccurrences(p1)`? In other words, how many times does the target `p1` appear in the bag? The answer depends on exactly how we implement `countOccurrences`, with these two possibilities:

1. `countOccurrences` could step through the elements of the bag, using the "==" operator to look for the target. In this case, we find one occurrence of the target `p1`, and `spots.countOccurrences(p1)` is 1.

2. `countOccurrences` could step through the elements of the bag, using `equals` to look for the target. In this case, both locations are equal to the target, and `spots.countOccurrences(p1)` is 2.

Every class has an `equals` method. For example, the `equals` method of the `String` class returns `true` if the two strings have the same sequence of characters. The `equals` method of the `Integer` wrapper class returns `true` if the two `Integer` objects hold the same `int` value. Because the `equals` method is always available, we'll use the second approach toward counting occurrences—and `spots.countOccurrences(p1)` is 2. Our bag documentation will make it clear that we count the occurrences of a non-null element by using its `equals` method.

> **Generic Collections Should Use the equals Method**
>
> When a generic collection tests for the presence of a non-null element, you should generally use the `equals` method rather than the "==" operator.

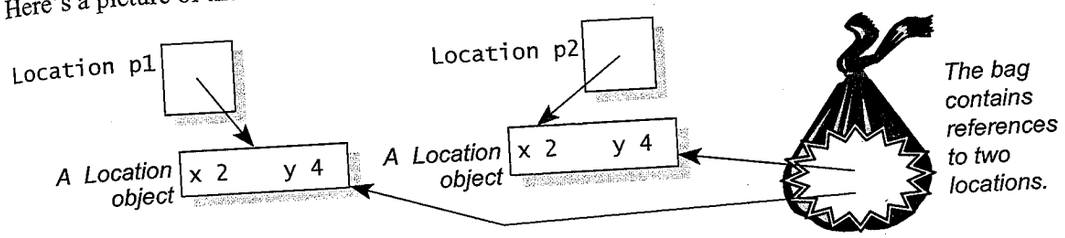
The Collection Is Really a Collection of References to Objects

There is another aspect of generic collections that you must understand. Although we use phrases such as "bag of objects," what we really have is a collection of *references* to objects. In other words, the bag does not contain separate copies of each object. Instead, the bag contains only references to each object that is added to the bag.

Let's draw some pictures to see what this means. Once again, we'll create two identical locations and put them in a bag:

```
ArrayBag<Location> spots = new ArrayBag<Location>( );
Location p1 = new Location(2, 4);  // x=2 and y=4
Location p2 = new Location(2, 4);  // Another at x=2 and y=4
spots.add(p1);
spots.add(p2);
```

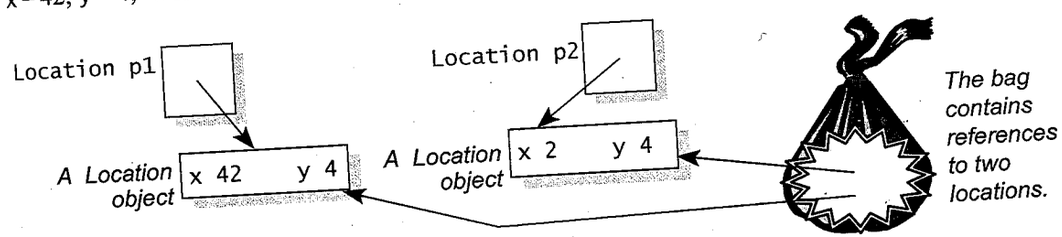
Here's a picture of the two locations and the bag after the five statements:



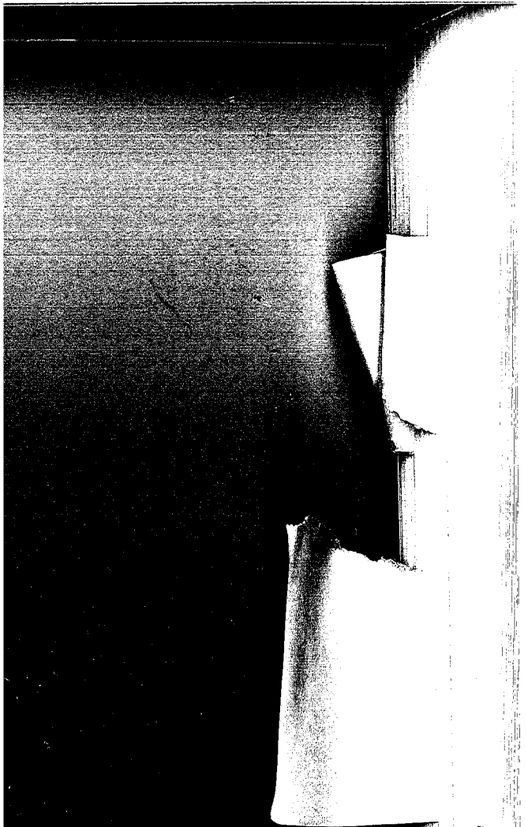
The references to the two locations are now in the bag. As we have already discussed, `spots.countOccurrences(p1)` is 2 because there are two locations with the coordinates `x=2, y=4`. What happens if we change `p1` or `p2`? For example, after putting the two locations in the bag, we could execute this statement:

```
p1.shift(40, 0);
```

This shifts `p1` by 40 units along the `x` axis and leaves `y` alone, so `p1` is now at `x=42, y=4`, as shown here:



The bag still has two locations, but one of those locations has shifted to `x=42`, `y=2`. Therefore, `spots.countOccurrences(p1)` is now just 1 (there is one location in the bag with the coordinates of `p1`).



Set Unused References to Null

Our bag uses a partially filled array. For example, a bag might contain 14 elements, but the array could be much larger than 14. When the bag contained primitive data, such as integers, we didn't really care what values were stored in the unused part of the array. If a bag stored 14 integers, then we didn't care what was beyond the first 14 locations of the array.

However, our `ArrayBag` no longer uses primitive types. Instead, we have a partially filled array of references to objects. To help Java collect unused memory, we should write our methods so that the unused part of the array contains only null references. For our `ArrayBag` class, this means that whenever we remove an element, we will assign null to the array location we are no longer using.

> **Set Unused Reference Variables to Null**
>
> When a collection class is converted to a generic collection class, make sure any unused reference variables are set to null. This usually occurs in methods that remove elements from the collection. Setting these variables to null will allow Java to collect any unused memory.

Eight Steps for Converting a Collection Class to a Generic Class

Here's a summary of exactly how we changed the `IntArrayBag` (from Section 3.2) into the generic `ArrayBag` class (in Figure 5.1 on page 260).

1. The Name of the Class. Each occurrence of the old class name, `IntArrayBag`, is changed to the new name, `ArrayBag<E>`. The name `ArrayBag` indicates that we have an array implemention of a bag, but we do not have any single underlying data type (such as `int`) because the new bag is a generic bag that holds an unspecified kind of element.

Notice that the name of the constructors is just `ArrayBag` (without the `<E>`).

2. The Type of the Underlying Element. Find all the spots where the old class used `int` to refer to an element in the bag. Change these spots to the generic type parameter, `E`. For example, the old bag had an array of integers as one of its instance variables. The new bag has an array of type `E`, as shown in these declarations of the new bag's instance variables:

```
public class ArrayBag implements Cloneable
{
   public E[ ] data;        // An array to store elements
   private int manyItems;   // How much of the array is used
   ...
```

Be careful because some `int` values do not refer to elements in the bag, and those must stay `int` (such as the `manyItems` instance variable).

3. Change static methods to generic static methods. The class may have some static methods. For example, the IntArrayBag had this static method:

```
public static IntArrayBag union(IntArrayBag b1, IntArrayBag b2)...
```

Any static method that depends on the generic type `E` must be changed to a generic method, which means that `<E>` appears just before the return type in the method's heading, like this for our `ArrayBag`:

```
public static <E> ArrayBag<E> union(ArrayBag<E> b1, ArrayBag<E> b2)...
```

4. Don't Create Any New E Objects or Arrays. As described on page 254, we cannot call any constructors or create arrays using the generic type parameter, `E`. So, the code to create an array of `E` objects actually creates an array of ordinary objects, and uses a typecast to assign this array to the `data` member variable:

```
data = (E[]) new Object[INITIAL_CAPACITY];
```

On the right hand side, we have created a new array that can contain references to any Java Object. The expression `(E[])` is a typecast that indicates that in this assignment statement, the programmer needs to treat the array as if it were an array of references to `E` objects, where `E` is the unknown type of the generic type parameter.

5. Equality Tests. Find all the spots where the old class used "`==`" or "`!=`" to compare two elements. Change these spots so that they use the `equals` method instead. For example, instead of `target != data[index]`, we will write `!target.equals(data[index])`.

6. Decide How to Treat the Null Reference. The new bag stores references to objects. You must decide whether to allow the null reference to be placed in the bag. For our bag, we'll allow the null reference to be placed in the bag, and we'll indicate this in the documentation. Some of the bag's methods will need special cases to deal with null references. For example, `countOccurrences` will search for a null target using "`==`" to count the number of times that null is in the bag, but a non-null target is counted by using its `equals` method.

7. Set Unused Reference Variables to Null. For our bag, this occurs in the `remove` method. Each time we remove an element, there is one array location that is no longer being used, and we set that array location to null (rather than letting it continue to refer to an object).

8. Update All Documentation. All documentation must be updated to show that the bag is a collection of references to objects.

Any collection class can be converted to a collection of objects by following these steps. For example, Programming Project 1 on page 302 asks you to convert the Sequence class from Section 3.3.

As for the new ArrayBag class, its specification and implementation are given in Figure 5.1. The changes are marked in the figure, and you'll also find one other improvement: We have included a grab method to allow a program to grab a randomly selected object from the bag. The implementation of the new grab method is similar to the grab method in the linked list version of the bag (see page 208 in Chapter 4).

Deep Clones for Collection Classes

The clone method creates a copy of an ArrayBag. As with other clone methods, adding or removing elements from the original will not affect the copy, nor vice versa. However, these elements are now references to objects; both the original and the copy contain references to the same underlying objects. Changing these underlying objects will affect both the original and the copy. An alternative cloning method, called **deep cloning**, can avoid the problem, as discussed in Programming Project 5 on page 302.

FIGURE 5.1 Specification and Implementation for the ArrayBag

Generic Class ArrayBag

❖ **public class ArrayBag<E> from the package edu.colorado.collections**

An ArrayBag<E> is a collection of references to E objects.

Limitations:

(1) The capacity of one of these bags can change after it's created, but the maximum capacity is limited by the amount of free memory on the machine. The constructors, add, clone, and union will result in an OutOfMemoryError when free memory is exhausted.

(2) A bag's capacity cannot exceed the largest integer 2,147,483,647 (Integer.MAX_VALUE). Any attempt to create a larger capacity results in failure due to an arithmetic overflow.

(3) Because of the slow linear algorithms of this class, large bags will have poor performance.

Specification

♦ **Constructor for the ArrayBag<E>**

public ArrayBag( )

Initialize an empty bag with an initial capacity of 10. Note that the add method works efficiently (without needing more memory) until this capacity is reached.

Postcondition:
This bag is empty and has an initial capacity of 10.

Throws: OutOfMemoryError
Indicates insufficient memory for: new Object[10].

(continued)

(FIGURE 5.1 continued)

♦ **Second Constructor for the ArrayBag<E>**

public ArrayBag(int initialCapacity)

Initialize an empty bag with a specified initial capacity. Note that the add method works efficiently (without needing more memory) until this capacity is reached.

Parameters:
initialCapacity – the initial capacity of this bag

Precondition:
initialCapacity is non-negative.

Postcondition:
This bag is empty and has the given initial capacity.

Throws: IllegalArgumentException
Indicates that initialCapacity is negative.

Throws: OutOfMemoryError
Indicates insufficient memory for new Object[initialCapacity].

♦ **add**

public void add(E element)

Add a new element to this bag. If this new element would take this bag beyond its current capacity, then the capacity is increased before adding the new element.

Parameters:
element – the new element that is being added

Postcondition:
A new copy of the element has been added to this bag.

Throws: OutOfMemoryError
Indicates insufficient memory for increasing the capacity.

Note:
Creating a bag with capacity beyond Integer.MAX_VALUE causes arithmetic overflow.

♦ **addAll**

public void addAll(ArrayBag<E> addend)

Add the contents of another bag to this bag.

Parameters:
addend – a bag whose contents will be added to this bag

Precondition:
The parameter, addend, is not null.

Postcondition:
The elements from addend have been added to this bag.

Throws: NullPointerException
Indicates that addend is null.

Throws: OutOfMemoryError
Indicates insufficient memory to increase the size of this bag.

Note:
Creating a bag with capacity beyond Integer.MAX_VALUE causes arithmetic overflow.

(continued)

(FIGURE 5.1 continued)

♦ **addMany**
`public void addMany(E... elements)`
Add a variable number of new elements to this bag. If these new elements would take this bag beyond its current capacity, then the capacity is increased before adding the new elements.

Parameters:
`elements` – a variable number of new elements that are all being added

Postcondition:
New copies of all the elements have been added to this bag.

Throws: `OutOfMemoryError`
Indicates insufficient memory for increasing the capacity.

Note:
Creating a bag with capacity beyond `Integer.MAX_VALUE` causes arithmetic overflow.

♦ **clone**
`public ArrayBag<E> clone( )`
Generate a copy of this bag.

Returns:
The return value is a copy of this bag. Subsequent changes to the copy will not affect the original, nor vice versa.

Throws: `OutOfMemoryError`
Indicates insufficient memory for creating the clone.

♦ **countOccurrences**
`public int countOccurrences(E target)`
Accessor method to count the number of occurrences of a particular element in this bag.

Parameters:
`target` – the reference to an `E` object to be counted

Returns:
The return value is the number of times that `target` occurs in this bag. If `target` is non-null, then the occurrences are found using the `target.equals` method.

♦ **ensureCapacity**
`public void ensureCapacity(int minimumCapacity)`
Change the current capacity of this bag.

Parameters:
`minimumCapacity` – the new capacity for this bag

Postcondition:
This bag's capacity has been changed to at least `minimumCapacity`. If the capacity was already at or greater than `minimumCapacity`, then the capacity is left unchanged.

Throws: `OutOfMemoryError`
Indicates insufficient memory for: `new Object[minimumCapacity]`.

(continued)

(FIGURE 5.1 continued)

♦ **getCapacity**
`public int getCapacity( )`
Accessor method to determine the current capacity of this bag. The add method works efficiently (without needing more memory) until this capacity is reached.

Returns:
the current capacity of this bag

♦ **grab**
`public E grab( )`
Accessor method to retrieve a random element from this bag.

Precondition:
This bag is not empty.

Returns:
a randomly selected element from this bag

Throws: `IllegalStateException`
Indicates that the bag is empty.

♦ **remove**
`public boolean remove(E target)`
Remove one copy of a specified element from this bag.

Parameters:
`target` – the element to remove from this bag

Postcondition:
If `target` was found in this bag, then one copy of `target` has been removed and the method returns `true`. Otherwise this bag remains unchanged, and the method returns `false`.

♦ **size**
`public int size( )`
Accessor method to determine the number of elements in this bag.

Returns:
the number of elements in this bag

♦ **trimToSize**
`public void trimToSize( )`
Reduce the current capacity of this bag to its size (i.e., the number of elements it contains).

Postcondition:
This bag's capacity has been changed to its current size.

Throws: `OutOfMemoryError`
Indicates insufficient memory for altering the capacity.

(continued)

(FIGURE 5.1 continued)

♦ **union**

`public static <E> ArrayBag<E> union(ArrayBag<E> b1, ArrayBag<E> b2)`

Create a new bag that contains all the elements from two other bags.

Parameters:
`b1` and `b2` – two bags

Precondition:
Neither `b1` nor `b2` is null.

Returns:
a new bag that is the union of `b1` and `b2`

Throws: `NullPointerException`
Indicates that one of the arguments is null.

Throws: `OutOfMemoryError`
Indicates insufficient memory for the new bag.

Note:
An attempt to create a bag with capacity beyond `Integer.MAX_VALUE` will cause the bag to fail with an arithmetic overflow.

Implementation

```
// File: ArrayBag.java from the package edu.colorado.collections
// Complete documentation is on pages 260-264 or from the ArrayBag link in:
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.collections;

public class ArrayBag<E> implements Cloneable
{
   // Invariant of the ArrayBag<E> generic class:
   //   1. The number of elements in the bag is in the instance variable manyItems.
   //   2. For an empty bag, we do not care what is stored in any of data;
   //      for a nonempty bag, the elements in the bag are stored in data[0]
   //      through data[manyItems-1], and we don't care what's in the rest of data.
   private E[ ] data;
   private int manyItems;

   public ArrayBag( )
   {
      final int INITIAL_CAPACITY = 10;
      manyItems = 0;
      data = (E[]) new Object[INITIAL_CAPACITY];
   }
```

The new bag stores an array of E objects rather than an array of int values.

(continued)

(FIGURE 5.1 continued)

```
   public ArrayBag(int initialCapacity)
   {
      if (initialCapacity < 0)
         throw new IllegalArgumentException
         ("initialCapacity is negative: " + initialCapacity);
      manyItems = 0;
      data = (E[]) new Object[initialCapacity];
   }

   public void add(E element)
   || The implementation is unchanged from the original in Figure 3.7 on page 134.

   public void addAll(ArrayBag<E> addend)
   || The implementation is unchanged from the original in Figure 3.7 on page 134.

   public void addMany(E... elements)
   || The implementation is unchanged from the original in Figure 3.7 on page 134.

   public ArrayBag<E> clone( )
   {  // Clone an ArrayBag<E>.
      ArrayBag<E> answer;

      try
      {
         answer = (ArrayBag<E>) super.clone( );
      }
      catch (CloneNotSupportedException e)
      {
         // This exception should not occur. But if it does, it would probably indicate a
         // programming error that made super.clone unavailable.
         // The most common error would be forgetting the "Implements Cloneable"
         // clause at the start of this class.
         throw new RuntimeException
         ("This class does not implement Cloneable");
      }

      answer.data = data.clone( );
      return answer;
   }
```

(continued)

(FIGURE 5.1 continued)

```
public int countOccurrences(E target)
{
   int answer;
   int index;

   answer = 0;

   if (target == null)
   {  // Count how many times null appears in the bag.
      for (index = 0; index < manyItems; index++)
         if (data[index] == null)
            answer++;
   }
   else
   {  // Use target.equals to determine how many times the target appears.
      for (index = 0; index < manyItems; index++)
         if (target.equals(data[index]))
            answer++;
   }

   return answer;
}
```

Special code is needed to handle the case where the target is null.

For a non-null target, we use target.equals instead of the "= =" operator.

```
public void ensureCapacity(int minimumCapacity)
{
   E biggerArray[ ];

   if (data.length < minimumCapacity)
   {
      biggerArray = (E[]) new Object[minimumCapacity];
      System.arraycopy(data, 0, biggerArray, 0, manyItems);
      data = biggerArray;
   }
}
```

```
public int getCapacity( )
```
|| The implementation is unchanged from the original in Figure 3.7 on page 137.

(continued)

(FIGURE 5.1 continued)

```
public E grab( )
{
   int i;

   if (manyItems == 0)
      throw new IllegalStateException("Bag size is zero.");

   i =  (int) (Math.random( ) * manyItems); // From 0 to manyItems-1
   return data[i];
}

public boolean remove(E target)
{
   int index; // The location of target in the data array

   // First, set index to the location of target in the data array.
   // If target is not in the array, then index will be set equal to manyItems.
   if (target == null)
   {  // Find the first occurrence of the null reference in the bag.
      index = 0;
      while ((index < manyItems) && (data[index] != null))
         index++;
   }
   else
   {  // Use target.equals to find the first occurrence of the target.
      index = 0;
      while ((index < manyItems) && (!target.equals(data[index])))
         index++;
   }

   if (index == manyItems)
      return false; // The target was not found, so nothing is removed.
   else
   {  // The target was found at data[index].
      manyItems--;
      data[index] = data[manyItems];
      data[manyItems] = null;
      return true;
   }

}
```

The unused array location is set to null to allow Java to collect the unused memory.

```
public int size( )
```
|| The implementation is unchanged from the original in Figure 3.7 on page 137.

(continued)

(FIGURE 5.1 continued)

```
   public void trimToSize( )
   {
      E trimmedArray[ ];

      if (data.length != manyItems)
      {
         trimmedArray = (E[]) new Object[manyItems];
         System.arraycopy(data, 0, trimmedArray, 0, manyItems);
         data = trimmedArray;
      }
   }

   public static <E> ArrayBag<E> union(ArrayBag<E> b1, ArrayBag<E> b2)
   {
      // If either b1 or b2 is null, then a NullPointerException is thrown.
      ArrayBag<E> answer =
         new ArrayBag<E>(b1.getCapacity( ) + b2.getCapacity( ));

      System.arraycopy(b1.data, 0, answer.data, 0, b1.manyItems);
      System.arraycopy(b2.data, 0, answer.data, b1.manyItems, b2.manyItems);
      answer.manyItems = b1.manyItems + b2.manyItems;
      return answer;
   }
}
```

Using the Bag of Objects

Using the bag of objects is easy. The program imports `edu.colorado.collections.ArrayBag`, and then a bag of objects can be declared. The same program may have several different bags for different purposes, or it may even have some of the original bags such as an `IntArrayBag`.

Figure 5.2 shows a demonstration program that uses three bags of strings. The program asks the user to type several adjectives and names. These words are placed in the bags, and then elements are grabbed out of the bags for the program to write a silly story called "Life."

FIGURE 5.2 Demonstration Program for the ArrayBag Generic Class

Java Application Program

```
// FILE: Author.java
// This program reads some words into bags. Then a silly story is written using these words.
import edu.colorado.collections.ArrayBag;
import java.util.Scanner;

public class Author
{
   private static Scanner stdin = new Scanner(System.in);

   public static void main(String[ ] args)
   {
      final int WORDS_PER_BAG = 4;  // Number of items per bag
      final int MANY_SENTENCES = 3; // Number of sentences in story

      ArrayBag<String> good  = new ArrayBag<String>(WORDS_PER_BAG);
      ArrayBag<String> bad   = new ArrayBag<String>(WORDS_PER_BAG);
      ArrayBag<String> names = new ArrayBag<String>(WORDS_PER_BAG);
      int line;

      // Fill the three bags with items typed by the program's user.
      System.out.println("Help me write a story.\n");
      getWords(good,  WORDS_PER_BAG, "adjectives that describe a good mood");
      getWords(bad,   WORDS_PER_BAG, "adjectives that describe a bad mood");
      getWords(names, WORDS_PER_BAG, "first names");
      System.out.println("Thank you for your kind assistance.\n");

      // Use the items to write a silly story.
      System.out.println("LIFE");
      System.out.println("by A. Computer\n");
      for (line = 1; line <= MANY_SENTENCES; line++)
      {
         System.out.print((String) names.grab( ));
         System.out.print(" was feeling ");
         System.out.print((String) bad.grab( ));
         System.out.print(", yet he/she was also ");
         System.out.print((String) good.grab( ));
         System.out.println(".");
      }
      System.out.println("Life is " + (String) bad.grab( ) + ".\n");
      System.out.println("The " + (String) good.grab( ) + " end.");
   }
```

The Scanner class is described in Appendix B.

(continued)

(FIGURE 5.2 continued)

```
    public static void getWords(ArrayBag<String> b, int n, String prompt)
    // Postcondition: The parameter, prompt, has been written as a prompt
    // to System.out. Then n strings have been read using stdin.next,
    // and these strings have been placed in the bag.
    {
       String userInput;
       int i;

       System.out.print("Please type " + n + " " + prompt);
       System.out.println(", separated by spaces.");
       System.out.println("Press the <return> key after the final entry:");
       for (i = 1; i <= n; i++)
       {
          userInput = stdin.next( );
          b.add(userInput);
       }
       System.out.println( );
    }
}
```

A Sample Dialogue

```
Help me write a story.

Please type 4 adjectives that describe a good mood, separated by spaces.
Press the <return> key after the final entry:
joyous happy lighthearted glad

Please type 4 adjectives that describe a bad mood, separated by spaces.
Press the <return> key after the final entry:
sad glum melancholy blue

Please type 4 first names, separated by spaces.
Press the <return> key after the final entry:
Michael Janet Tim Hannah

Thank you for your kind assistance.

LIFE by A. Computer

Tim was feeling glum, yet he/she was also glad.
Michael was feeling melancholy, yet he/she was also joyous.
Hannah was feeling blue, yet he/she was also joyous.
Life is blue.

The lighthearted end.
```

Details of the Story-Writing Program

The bags are declared in the demonstration program as you would expect:

```
ArrayBag<String> good  = new ArrayBag<String>(WORDS_PER_BAG);
ArrayBag<String> bad;  = new ArrayBag<String>(WORDS_PER_BAG);
ArrayBag<String> names; = new ArrayBag<String>(WORDS_PER_BAG);
```

The number, `WORDS_PER_BAG`, is 4 in the story program, so each of these bags has an initial capacity of 4. To get the actual words from the user, the program calls `getWords`, with this heading:

```
public static void getWords(ArrayBag<String> b, int n, String prompt)
```

The method uses the third parameter, `prompt`, as part of a prompt that asks the user to type `n` words. For example, if `prompt` is the string constant `"first names"` and `n` is 4, then the `getWords` method writes this prompt:

```
Please type 4 first names, separated by spaces.
Press the <return> key after the final entry:
```

The method then reads `n` strings by using the `Scanner` class (described in Appendix B).

As the strings are read, the `getWords` method places them in the bag by activating `b.add`. In all, the main program activates `getWords` three times to get three different kinds of strings. The literary merit of the program's story is debatable, but the ability to use bags of objects is clearly important.

Self-Test Exercises for Section 5.3

13. We converted an `IntArrayBag` to a generic `ArrayBag<E>`. During the conversion, does every occurrence of `int` get changed to `E`?
14. Our bag is a bag of references to objects. Can the null reference be placed into our bag?
15. Write some code that declares a bag of Integers and then puts the numbers 1 through 10 into the bag. The numbers are objects of the wrapper class, `Integer`, not the primitive class `int`.
16. The original `countOccurrences` method tested for the occurrence of a target by using the boolean expression `target == data[index]`. What different boolean expression is used in the `countOccurrences` method of the bag of objects?
17. What technique did the story-writing program use to read strings from the keyboard?

18. Consider this code that puts a Location into a bag, changes the name, and then tests whether the original name is in the bag. The Location class is from Figure 2.5 on page 67. What does the code print?

```
ArrayBag<Location> points = new ArrayBag<Location>( );
Location origin = new Location(0,0);
Location moving = new Location(0,0);
points.add(moving);
moving.shift(5,10);
System.out.println(points.countOccurrences(origin));
System.out.println(points.countOccurrences(moving));
```

5.4 GENERIC NODES

Nodes That Contain Object Data

Our node class from Chapter 4 can also be converted to a generic class. In the conversion, the new class, called Node, allows us to build linked lists of nodes in which the type of data in each node is determined by the generic type parameter. Here is a comparison of the original IntNode declaration with our new generic class:

Original IntNode:

```
public class IntNode
{
    private int data;
    IntNode link;

    The methods use
    the int data.
}
```

Generic Node Class:

```
public class <E> Node
{
    private E data;
    Node<E> link;

    The methods use
    the E data.
}
```

With the new Node class, each node contains a piece of data that is a reference to an E object, but we don't specify exactly what E is. The implementation of the methods is based on the IntNode methods, following the same steps that we used before on page 258. The resulting Node class is given in Appendix E. You may find yourself using it beyond this book, too.

PITFALL

MISUSE OF THE EQUALS METHOD

When you convert a collection class to contain objects, it is tempting to blindly change every occurrence of the "==" operator to use the equals method instead. There are two pitfalls to beware of.

First, beware of null references. You cannot activate the equals method of a null reference. For example, here is the implementation of the new listSearch

method from the generic Node class. The method searches the linked list for an occurrence of a particular target, and returns a reference to the node that contains the target. If no such node is found, then the null reference is returned:

```
public static <E> Node<E> listSearch(Node<E> head, E target)
{
   Node<E> cursor;

   if (target == null)
   {  // Search for a node in which the data is the null reference.
      for (cursor = head; cursor != null; cursor = cursor.link)
         if (cursor.data == null)
            return cursor;
   }
   else
   {  // Search for a node that contains the non-null target.
      for (cursor = head; cursor != null; cursor = cursor.link)
         if (target.equals(cursor.data))
            return cursor;
   }

   return null;
}
```

The target may be the null reference—in that case, we're searching for a node in which the data is null. This search is carried out in the first part of the large if-statement. We test whether a particular node contains null data with the boolean expression `cursor.data == null`. On the other hand, for a non-null target, the search is carried out in the else-part of the large if-statement. We test whether a particular node contains a non-null target with the boolean expression `target.equals(cursor.data)`.

In general, the equals method may be used only when you know that the target is non-null.

The second thing to beware of: You should change an equality test "==" to the equals method only where the program is comparing data. Don't change other comparisons. For example, here is the listPart method of the generic Node class:

```
public static <E> Node<E>[ ] listPart(Node<E> start, Node<E> end)
{
   Node<E> copyHead;
   Node<E> copyTail;
   Node<E> cursor;
   Node<E>[ ] answer = new (Node<E>[]) Object[2];

   // Check for illegal null at start or end.
   if (start == null)
```