CHAPTER 3

Collection Classes

(I am large. I contain multitudes.)

WALT WHITMAN
Song of Myself

The `Throttle` and `Location` classes in Chapter 2 are good examples of abstract data types. But their applicability is limited to a few specialized programs. This chapter begins the presentation of several ADTs with broad applicability to programs large and small. The ADTs in this chapter—bags and sequences—are small, but they provide the basis for more complex ADTs. The chapter also includes information about how to write a test program for a class.

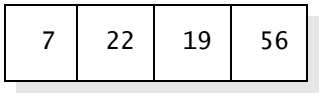
an ADT in which each object contains a collection of elements

All of the ADTs in this chapter are examples of **collection classes**. Intuitively, a collection class is a class where each object contains a collection of elements. For example, one program might keep track of a collection of integers, perhaps the collection of test scores for a group of students. Another program, perhaps a cryptography program, can use a collection of characters.

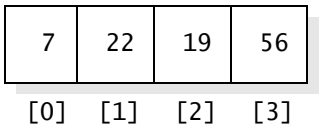
There are many different ways to implement a collection class; the simplest approach utilizes an array, so this chapter begins with a quick review of Java arrays before approaching actual collection classes.

3.1 A REVIEW OF JAVA ARRAYS

An array is a sequence with a certain number of components. We draw arrays with each component in a separate box. For example, here's an array of the four integers 7, 22, 19, and 56:



Each component of an array can be accessed through an index. In Java, the indexes are written with square brackets, beginning with `[0]`, `[1]`,.... The array shown has four components, so the indexes are [0] through [3], as shown here:



In these examples, each component is an integer, but arrays can be built for any fixed data type: arrays of double numbers, arrays of boolean values, even arrays where the components are objects from a new class that you write yourself.

An array is declared like any other variable, except that a pair of square brackets is placed after the name of the data type. For example, a program can declare an array of integers like this:

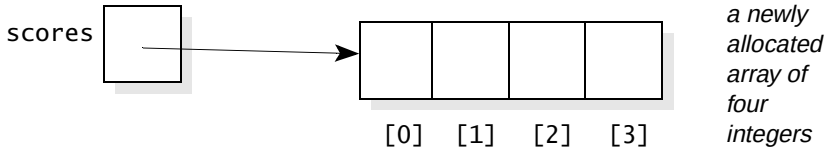
```
int[ ] scores;
```

The name of this array is `scores`. The components of this array are integers, but as we have mentioned, the components may be any fixed type. For example, an array of double numbers would use `double[ ]` instead of `int[ ]`.

An array variable, such as `scores`, is capable of referring to an array of any size. In fact, an array variable is a reference variable, just like the reference variables we have used for other objects, and arrays are created with the same `new` operator that allocates other objects. For example, we can write these statements:

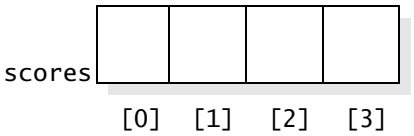
```
int[ ] scores;
scores = new int[4];
```

The number `[4]`, occurring with the `new` operator, indicates that we want a new array with four components. Once both statements finish, `scores` refers to an array with four integer components, as shown here:



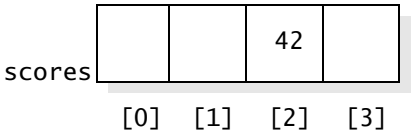
This is an accurate picture, showing how `scores` refers to a new array of four integers, but the picture has some clutter that we can usually omit. Here is a

simpler picture that we'll usually use to show that `scores` refers to an array of four integers:



Both pictures mean the same thing. The first picture is a more accurate depiction of what Java actually does; the second is a kind of shorthand which is typical of what programmers draw to illustrate an array.

Once an array has been allocated, individual components can be selected using the square bracket notation with an index. For example, with scores allocated as shown, we can set its [2] component to 42 with the assignment `scores[2] = 42`. The result is shown here:



Pitfall: Exceptions That Arise from Arrays

PITFALL

Two kinds of exceptions commonly arise from programming errors with arrays. One problem is to try to use an array variable before the array has been allocated. For example, suppose we declare `int[ ] scores`, but we forget to use the `new` operator to create an array for `scores` to refer to. At this point, `scores` is actually a reference variable, just like the reference variables that we discussed for other kinds of objects on page 50. But merely declaring a reference variable does not allocate an array, and it is a programming error to try to access a component such as `scores[2]`. A program that tries to access a component of a nonexistent array may throw a `NullPointerException` (if the reference is null) or there may be a compile-time error (if the variable is an uninitialized local variable).

A second common programming error is trying to access an array outside of its bounds. For example, suppose that `scores` refers to an array with four components. The indexes are `[0]` through `[3]`, so it is an error to use an index that is too small (such as `scores[-1]`) or too large (such as `scores[4]`). A program that tries to use these indexes will throw an `ArrayIndexOutOfBoundsException`.

The Length of an Array

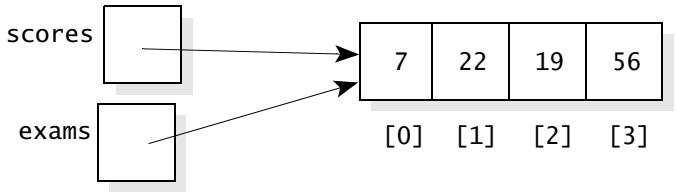
Every array has an instance variable called `length`, which tells the number of components in the array. For example, consider `scores = new int[4]`. After this allocation, `scores.length` is 4. Notice that `length` is not a method, so the syntax is merely `scores.length` (with no argument list). By the way, if an array variable is the null reference, then you cannot ask for its length (trying to do so results in a `NullPointerException`).

Assignment Statements with Arrays

A program can use an assignment statement to make two array variables refer to the same array. Here's some example code:

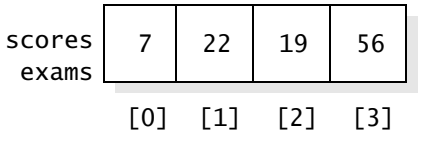
```
int[ ] scores;
int[ ] exams;

scores = new int[4];
scores[0] = 7;
scores[1] = 22;
scores[2] = 19;
scores[4] = 56;
exams = scores;
```

After these statements, `scores` refers to an array containing the four integers 7, 22, 19, and 56. The assignment statement, `exams = scores`, causes `exams` to refer to the exact same array. Here is an accurate drawing of the situation:



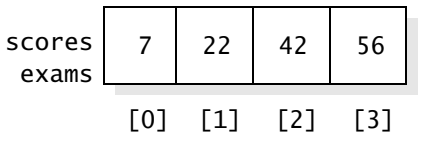
after the assignment, both array variables refer to the same array

Here's a shorthand drawing of the same situation to show that `scores` and `exams` refer to the same array:



after the assignment, both array variables refer to the same array

In this example, there is only one array, and both array variables refer to this one array. Any change to the array will affect both `scores` and `exams`. For example, after the above statements we might assign `exams[2] = 42`. The situation after the assignment to `exams[2]` is shown here:



the [2] component has been changed to 42

At this point, both `exams[2]` and `scores[2]` are 42.

Clones of Arrays

In Chapter 2 you saw how to use a `clone` method to create a completely separate copy of an object. Every Java array comes equipped with a `clone` method to create a copy of the array. Just like the other clones that you've seen, changes to the original array don't affect the clone, and changes to the clone don't affect the original array. Here's an example:

```
int[ ] scores;
int[ ] exams;

scores = new int[4];
scores[0] = 7;
scores[1] = 22;
scores[2] = 19;
scores[3] = 56;
exams = (int[ ]) scores.clone( );
```

The final statement in this example uses `scores.clone( )` to create a copy of the `scores` array. The data type of the return value of any `clone` method is Java's `Object` data type and not an array. Because of this, we usually cannot use the `clone` return value directly. For example, we cannot write an assignment:

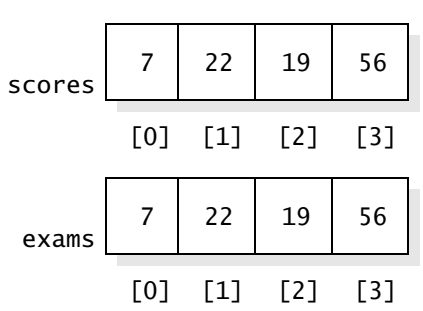
```
exams = scores.clone( );
```
this has a compile-time error

Instead, we must apply a typecast to the `clone` return value, converting it to an integer array before we assign it to exams, like this:

```
exams = (int[ ]) scores.clone( );
```

The expression `(int[ ])` tells the compiler to treat the return value of the clone method as an integer array.

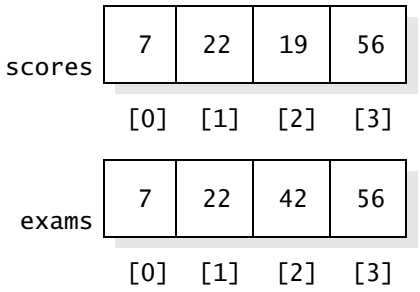
After the assignment statement, `exams` refers to a new array which is an exact copy of the `scores` array, as shown here:



after creating the clone, there are two separate arrays

There are now two separate arrays. Changes to one array do not affect the other. For example, after the above statements we might assign `exams[2] = 42`.

The situation after the assignment to `exams[2]` is shown here:



exams[2] has changed to 42, but scores[2] is unchanged

At this point, `exams[2]` is 42, but `scores[2]` is unchanged.

Array Parameters

An array can be a parameter to a method. Here's an example method with an array as a parameter:

- **put42s**
  `public static void put42s(int[ ] data)`
  Put 42 in every component of an array.
  Parameters:
  `data` – an array of integers
  Postcondition:
  All components of `data` have been set to 42.

```
public static void put42s(int[ ] data)
{
   int i;
   for (i = 0; i < data.length; i++)
      data[i] = 42;
}
```

Perhaps this is a silly example (when was the last time you really wanted to put 42 in *every* component of an array?), but the example is a good way to show how an array works as a parameter. Notice how the array parameter is indicated by placing array brackets after the parameter name. In the `put42s` example, the array is called `data`, so the parameter list is `(int[ ] data)`.

When a method is activated with an array parameter, the parameter is initialized to refer to the same array that the actual argument refers to. Therefore, if the method changes the components of the array, the changes do affect the actual argument. For example, this code activates the `put42s` method:

```
int[ ] example = new int[7];
put42s(example);
```

After these statements, all seven components of the `example` array contain 42.

Array Parameters

When a parameter is an array, then the parameter is initialized to refer to the same array that the actual argument refers to. Therefore, if the method changes the components of the array, the changes do affect the actual argument.

Self-Test Exercises

1. Write code that follows these steps: (1) Declare an integer array variable called `b`; (2) Allocate a new array of 1000 integers for `b` to refer to; and (3) Place the numbers 1 through 1000 in the array.
2. Write a Java expression that will indicate how many elements are in the array `b` (from the previous exercise).
3. What is the output from this code:
```
int[ ] a, b;
a = new int[10];
a[5] = 0;
b = a;
a[5] = 42;
System.out(b[5]);
```
4. What is the output from this code:
```
int[ ] a, b;
a = new int[10];
a[5] = 0;
b = (int [ ]) a.clone( );
a[5] = 42;
System.out(b[5]);
```
5. Suppose that an array is passed as a parameter to a method, and the method changes the first component of the array to 42. What effect does this have on the actual argument back in the calling program?
6. Write a method that copies *n* elements from the front of one integer array to the front of another. The two arrays and the number *n* are all arguments to the method. Include a precondition/postcondition contract as part of your implementation.

3.2 AN ADT FOR A BAG OF INTEGERS

This section provides an example of a collection class. In this first example, the collection class will use an array to store its collection of elements (but later we will see other ways to store collections).

The example collection class is called a *bag of integers*. To describe the bag data type, think about an actual bag—a grocery bag or a garbage bag—and

imagine writing integers on slips of paper and putting them in the bag. A **bag of integers** is similar to this imaginary bag: a container that holds a collection of integers that we place into it. A bag of integers can be used by any program that needs to store a collection of integers for its own use. For example, later we will write a program that keeps track of the ages of your family's members. If you have a large family with ten people, the program keeps track of ten ages—and these ages are kept in a bag of integers.

The Bag ADT—Specification

We've given an intuitive description of a bag of integers. We will implement this bag as a class called `IntArrayBag`, in which the integers are stored in an array. In general, we'll use a three-part name for a collection: "`Int`" specifies the type of the elements in the bag; "`Array`" indicates the mechanism for storing the elements; and "`Bag`" indicates the kind of collection. For a precise specification of the `IntArrayBag` class, we must describe each of the public methods to manipulate an `IntArrayBag` object. These descriptions will later become our specifications, including a precondition/postcondition contract for each method. Let's look at the methods one at a time.

The Constructors. The `IntArrayBag` class has two constructors to initialize a new, empty bag. One constructor has a parameter, as shown in this heading:

```
public IntArrayBag(initialCapacity)
```

The parameter, `initialCapacity`, is the initial capacity of the bag—the number of elements that the bag can hold. Once this capacity is reached, more elements can still be added and the capacity will automatically increase in a manner that you'll see in a moment.

The other constructor has no parameters, and it constructs a bag with an initial capacity of ten.

The add Method. This is a modification method that places a new integer, called `element`, into a bag. Here is the heading:

```
public void add(int element)
```

As an example, here are some statements for a bag called `firstBag`:

```
IntArrayBag firstBag = new IntArrayBag( );
firstBag.add(8);
firstBag.add(4);          After these statements, firstBag
firstBag.add(8);          contains two 8s and a 4.
```

After these statements are executed, `firstBag` contains three integers: the number 4 and two copies of the number 8. It is important to realize that a bag can contain many copies of the same integer, such as this example, which has two copies of 8.

The `remove` Method. This modification method removes a particular number from a bag. The heading is shown here:

```
public boolean remove(int target)
```

Provided that `target` is actually in the bag, the method removes one copy of `target` and returns `true` to indicate that something has been removed. If `target` is not in the bag, then the method returns `false` without changing the bag.

The `size` Method. This accessor method returns the count of how many integers are in a bag. The heading for the `size` method is:

```
public int size( )
```

For example, suppose `firstBag` contains one copy of the number 4 and two copies of the number 8. Then `firstBag.size( )` returns 3.

The `countOccurrences` Method. This is an accessor method that determines how many copies of a *particular* number are in a bag. The heading is:

```
public int countOccurrences(int target)
```

The activation of `countOccurrences(n)` returns the number of occurrences of `n` in a bag. For example, if `firstBag` contains the number 4 and two copies of the number 8, then we will have these values:

```
System.out.println(firstBag.countOccurrences(1));
System.out.println(firstBag.countOccurrences(4));
Systme.out.println(firstBag.countOccurrences(8));
```

Prints 0

Prints 1

Prints 2

The `addAll` Method. The `addAll` method allows us to add the contents of one bag to the existing contents of another bag. The method has this heading:

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

This is an interesting method for the `IntArrayBag` class because the parameter is also an `IntArrayBag` object. We use the name `addend` for the parameter, meaning "something to be added." As an example, suppose we create two bags called `helter` and `skelter`, and we then want to add all the contents of `skelter` to `helter`. This is done with `helter.addAll(skelter)`, as shown here:

```
IntArrayBag helter= new IntArrayBag( );
IntArrayBag skelter = new IntArrayBag( );
helter.add(8);
skelter.add(4);
skelter.add(8);
helter.addAll(skelter);
```

This adds the contents of skelter to what's already in helter.

After these statements, `helter` contains one 4 and two 8s.

The `union` Method. The **union** of two bags is a new larger bag that contains all the numbers in the first bag and all the numbers in the second bag, as shown here:

We will implement `union` with a static method that has the two parameters shown here:

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

The `union` method computes the union of `b1` and `b2`. For example:

```
IntArrayBag part1 = new IntArrayBag( );
IntArrayBag part2 = new IntArrayBag( );

part1.add(8);
part1.add(9);
part2.add(4);
part2.add(8);
```

This computes the union of the two bags, putting the result in a third bag.

```
IntArrayBag total = IntArrayBag.union(part1, part2);
```

After these statements, `total` contains one 4, two 8s, and one 9.

The `union` method is similar to `addAll`, but the usage is different. The `addAll` method is an ordinary method that is activated by a bag, for example `helter.addAll(skelter)`, which adds the contents of `skelter` to `helter`. On the other hand, union is a static method with two arguments. As a static method, `union` is not activated by any one bag. Instead, the activation of `IntArrayBag.union(part1, part2)` creates and returns a new bag that includes the contents of both `part1` and `part2`.

The `clone` Method. As part of our specification, we require that bag objects can be copied with a clone method. For example:

b now contains a 42.

```
IntArrayBag b = new IntArrayBag( );
b.add(42);
IntArrayBag c = (IntArrayBag) b.clone( );
```

c is initialized as a clone of b.

At this point, because we are only specifying which operations can manipulate a bag, we don't need to say anything more about the `clone` method.

Three Methods That Deal with Capacity. Each bag has a current **capacity**, which is the number of elements the bag can hold without having to request more memory. Once the capacity is reached, more elements can still be added by the `add` method. In this case, the `add` method itself will increase the capacity as needed. In fact, our implementation of `add` will double the capacity whenever the bag becomes full.

With this in mind, you might wonder why a programmer needs to worry about the capacity at all. For example, why does the constructor require the programmer to specify an initial capacity? Couldn't we always use the constructor that has an initial capacity of ten and have the `add` method increase capacity as more and more elements are added? Yes, this approach will always work correctly. But if there are many elements, then many of the activations of `add` would need to increase the capacity. This could be inefficient—in fact, increasing the capacity will be the least efficient operation of the entire bag. To avoid repeatedly increasing the capacity, a programmer provides an initial guess at the needed capacity for the constructor.

For example, suppose a programmer expects no more than 1000 elements for a bag named `kilosack`. The bag is declared this way, with an initial capacity of 1000:

```
IntArrayBag kilosack = new IntArrayBag(1000);
```

After this declaration, the programmer can place 1000 elements in the bag without worrying about the capacity. Later, the programmer can add more elements to the bag, maybe even more than 1000. If there are more than 1000 elements, then `add` increases the capacity as needed.

There are three methods that allow a programmer to manipulate a bag's capacity after the bag is in use. The methods have these headers:

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

The first method, `getCapacity`, just returns the current capacity of the bag. The second method, `ensureCapacity`, increases the capacity to a specified minimum amount. For example, in order to ensure that a bag called `bigboy` has a capacity of at least 10,000, we would activate `bigboy.ensureCapacity(10000)`.

The third method, `trimToSize`, reduces the capacity of a bag to its current size. For example, suppose that `bigboy` has a current capacity of 10,000, but it contains only 42 elements and we are not planning to add any more. Then we can reduce the current capacity to 42 with the activation `bigboy.trimToSize( )`. Trimming the capacity is never required, but doing so can reduce the memory used by a program.

That's all the methods, and we're almost ready to write the methods' specifications. But first, there are some limitations that we'd like to discuss.

`OutOfMemoryError` and Other Limitations for Collection Classes

Our plan is to store a bag's elements in an array, and to increase the capacity of the array as needed. The memory for any array comes from a location called the program's **heap** (also called the **free store**). In fact, the memory for all Java objects comes from the heap. Some computers provide huge heaps—Java implementations running on a machine with a "64-bit address space" have the potential for more than 10^{18} integers. But even the largest heap can be exhausted by creating large arrays or other objects.

what happens when the heap runs out of memory?

If a heap has insufficient memory for a new object or array, then the result is a Java exception called `OutOfMemoryError`. This exception is thrown automatically by an unsuccessful "new" operation. For example, if there is insufficient memory for a new `Throttle` object, then `Throttle t = new Throttle( )` throws an `OutOfMemoryError`. Experienced programmers may monitor the size of the heap and the amount that is still unused. Our programs won't attempt such monitoring, but our specification for any collection class will always mention that the maximum capacity is limited by the amount of free memory. To aid more experienced programmers, the specification will also indicate precisely which methods have the possibility of throwing an `OutOfMemoryError`. (Any method that uses the "new" operation could throw this exception.)

collection classes may be limited by the maximum value of an integer

Many collection classes have another limitation that is tied to the maximum value of an integer. In particular, our bag stores the elements in an array, and every array has integers for its indexes. Java integers are limited to no more than 2,147,483,647, which is also written as `Integer.MAX_VALUE`. An attempt to create an array with a size beyond `Integer.MAX_VALUE` results in an arithmetic overflow during the calculation of the size of the array. Such an overflow usually produces an array size that Java's runtime system sees as negative. This is because Java represents integers so that the "next" number after `Integer.MAX_VALUE` is actually the smallest negative number.

Programmers often ignore the array-size overflow problem (since today's machines generally have an `OutOfMemoryError` before `Integer.MAX_VALUE` is approached). We won't provide special code to handle this problem, but we won't totally ignore the problem either. Instead, our documentation will indicate precisely which methods have the potential for an array-size overflow. We'll also add a note to advise that large bags should probably use a different implementation method anyway, because many of the array-based algorithms are slow for large bags ($O(n)$, where n is the number of elements in the bag).

The `IntArrayBag` Class—Specification

We now know enough about the bag to write a specification, as shown in Figure 3.1. We've used the name `IntArrayBag` for this class, and it is the first class of a package named `edu.colorado.collections`.

The specification also states the bag's limitations: the possibility of an `OutOfMemoryError` (when the heap is exhausted), a note about the limitation of the capacity, and a note indicating that large bags will have poor performance.

FIGURE 3.1 Specification for the `IntArrayBag` Class

Class IntArrayBag

❖ **public class IntArrayBag from the package edu.colorado.collections**

An `IntArrayBag` is a collection of `int` numbers.

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 constructor, `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 IntArrayBag**

`public IntArrayBag( )`

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 int[10]`.

◆ **Second Constructor for the IntArrayBag**

`public IntArrayBag(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 int[initialCapacity]`.

(continued)

(FIGURE 3.1 continued)

- **add**

  `public void add(int 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:
  An attempt to increase the capacity beyond `Integer.MAX_VALUE` will cause this bag to fail with an arithmetic overflow.

- **addAll**

  `public void addAll(IntArrayBag 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:
  An attempt to increase the capacity beyond `Integer.MAX_VALUE` will cause this bag to fail with an arithmetic overflow.

- **clone**

  `public Object 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. The return value must be typecast to an `IntArrayBag` before it is used.

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

(continued)

(FIGURE 3.1 continued)

- **countOccurrences**

  `public int countOccurrences(int target)`

  Accessor method to count the number of occurrences of a particular element in this bag.

  Parameters:
  `target` – the element that needs to be counted

  Returns:
  the number of times that `target` occurs in this bag

- **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 int[minimumCapacity]`.

- **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

- **remove**

  `public boolean remove(int 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

(continued)

(FIGURE 3.1 continued)

- **trimToSize**

  `public void trimToSize( )`

  Reduce the current capacity of this bag to its actual 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.

- **union**

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

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

  Parameters:
  `b1` – the first of two bags
  `b2` – the second of 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.

The `IntArrayBag` Class—Demonstration Program

With the specification in hand, we can write a program that uses a bag. We don't need to know what the instance variables of a bag are, and we don't need to know how the methods are implemented. As an example, a demonstration program appears in Figure 3.2. The program asks a user about the ages of family members. The user enters the ages followed by a negative number to indicate the end of the input. (Using a special value to end a list is a common technique called a **sentinel value**.) A typical dialogue with the program looks like this:

```
Type the ages of your family members.
Type a negative number at the end and press return.
5   19    47   -1
Type those ages again. Press return after each age.
Age: 19
Yes, I've got that age and will remove it.
```

```
Age: 36
No, that age does not occur!
Age: 5
Yes, I've got that age and will remove it.
Age: 47
Yes, I've got that age and will remove it.
May your family live long and prosper.
```

the EasyReader class from Appendix B allows simple kinds of input

The program puts the ages in a bag and then asks the user to type the ages again. The program's interaction with the user is handled through a class called `EasyReader`, which contains various simple input methods. The class is fully described in Appendix B, but for this program all that's needed is a single `EasyReader` called `stdin`, which is attached to standard input (`System.in`).

Once `stdin` is set up, an integer can be read with either of two methods: (1) `stdin.intInput` (which simply reads an integer input), or (2) `stdin.intQuery` (which prints a prompt and then reads an integer input). You may find the `EasyReader` class useful for your own demonstration programs.

As for the `IntArrayBag` class itself, we still don't know how the implementation will work, but we're getting there.

FIGURE 3.2 Demonstration Program for the Bag Class

Java Application Program

```
// FILE: BagDemonstration.java
// This small demonstration program shows how to use the IntArrayBag class
// from the edu.colorado.collections package.

import edu.colorado.collections.IntArrayBag;
import edu.colorado.io.EasyReader;

class BagDemonstration
{
   private static EasyReader stdin = new EasyReader(System.in);

   public static void main(String[ ] args)
   {
      IntArrayBag ages = new IntArrayBag( );
      getAges(ages);
      checkAges(ages);
      System.out.println("May your family live long and prosper.");
   }
```

the EasyReader class is described in Appendix B

(continued)

(FIGURE 3.2 continued)

```
   public static void getAges(IntArrayBag ages)
   // The getAges method prompts the user to type in the ages of family members. These
   // ages are read and placed in the ages bag, stopping when the user types a negative
   // number. This demonstration does not worry about the possibility of running out
   // of memory (therefore, an OutOfMemoryError is possible).
   {
      int userInput; // An age from the user's family

      System.out.println("Type the ages of your family members.");
      System.out.println("Type a negative number at the end and press return.");
      userInput = stdin.intInput( );
      while (userInput >= 0)
      {
         ages.add(userInput);
         userInput = stdin.intInput( );
      }
   }

   public static void checkAges(IntArrayBag ages)
   // The checkAges method prompts the user to type in the ages of family members once
   // again. Each age is removed from the ages bag when it is typed, stopping when the bag
   // is empty.
   public static void checkAges(IntArrayBag ages)
   {
      int userInput; // An age from the user's family

      System.out.print("Type those ages again. ");
      System.out.println("Press return after each age.");
      while (ages.size( ) > 0)
      {
         userInput = stdin.intQuery("Next age:  ");
         if (ages.countOccurrences(userInput) == 0)
            System.out.println("No, that age does not occur!");
         else
         {
            System.out.println("Yes, I've got that age and will remove it.");
            ages.remove(userInput);
         }

      }
   }

}
```

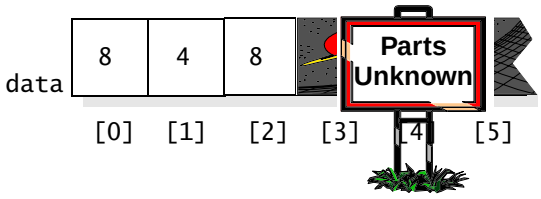
The `IntArrayBag` Class—Design

There are several ways to design the `IntArrayBag` class. For now, we'll keep things simple and design a somewhat inefficient data structure using an array. The data structure will be redesigned several times to obtain more efficiency.

We start the design by thinking about the data structure—the actual configuration of private instance variables used to implement the class. The primary structure for our design is an array that stores the elements of a bag. Or, to be more precise, we use the *beginning* part of a large array. Such an array is called a **partially filled array**. For example, if the bag contains the integer 4 and two copies of 8, then the first part of the array could look this way:

use the beginning part of an array

Components of the partially filled array contain the elements of the bag.



This array, called `data`, will be one of the private instance variables of the `IntArrayBag` class. The length of the array will be determined by the current capacity, but as the picture indicates, when we are using the array to store a bag with just three elements, we don't care what appears beyond the first three components. Starting at index 3, the array might contain all zeros, or it might contain garbage, or our favorite number—it really doesn't matter.

Because part of the array can contain garbage, the `IntArrayBag` class must keep track of one other item: *How much of the array is currently being used?* For example, in the previous picture, we are using only the first three components of the array because the bag contains three elements. The amount of the array being used can be as small as zero (an empty bag) or as large as the current capacity. The amount increases as elements are added to the bag, and it decreases as elements are removed. In any case, we will keep track of the amount in a private instance variable called `manyItems`. With this approach, there are two instance variables for a bag:

the bag's instance variables

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

    || The public methods will be given in a moment.
}
```

Notice that we are planning to implement a `clone` method, therefore we indicate "`implements Cloneable`" at the start of the class definition.

The Invariant of an ADT

We've defined the bag data structure, and we have a good intuitive idea of how the structure will be used to represent a bag of elements. But as an aid in implementing the class we should also write down an explicit statement of how the data structure is used to represent a bag. In the case of the bag, we need to state how the instance variables of the class are used to represent a bag of elements. There are two rules for our bag implementation:

rules that dictate how the instance variables are used to represent a value

1. The number of elements in the bag is stored 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 of the bag are stored in `data[0]` through `data[manyItems-1]`, and we don't care what is stored in the rest of `data`.

The rules that dictate how the instance variables of a class represent a value (such as a bag of elements) are called the **invariant of the ADT**. The knowledge of these rules is essential to the correct implementation of the ADT's methods. With the exception of the constructors, each method depends on the invariant being valid when the method is activated. And each method, including the constructors, has a responsibility of ensuring that the invariant is valid when the method finishes. In some sense, the invariant of an ADT is a condition that is an *implicit* part of every method's postcondition. And (except for the constructors) it is also an implicit part of every method's precondition. The invariant is not usually written as an *explicit* part of the precondition and postcondition because the programmer who uses the ADT does not need to know about these conditions. But to the implementor of the ADT, the invariant is indispensable. In other words, the invariant is a critical part of the implementation of an ADT, but it has no effect on the way the ADT is used.

Key Design Concept

The invariant is a critical part of an ADT's implementation.

The Invariant of an ADT

When you design a new class, always make an explicit statement of the rules that dictate how the instance variables are used. These rules are called the **invariant of the ADT**. All of the methods (except the constructors) can count on the invariant being valid when the method is called. Each method also has the responsibility of ensuring that the invariant is valid when the method finishes.

Once the invariant of an ADT is stated, the implementation of the methods is relatively simple because there is no interaction between the methods—except for their cooperation at keeping the invariant valid. We'll look at these implementations one at a time, starting with the constructors.

The `IntArrayBag` ADT—Implementation

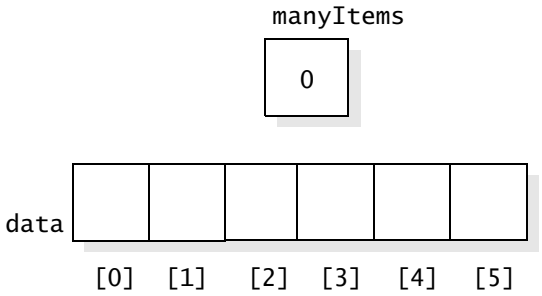
The Constructor. Every constructor has one primary job: to set up the instance variables correctly. In the case of the bag, the constructor must set up the instance variables so that they represent an empty bag with a current capacity given by the parameter `initialCapacity`. The bag has two instance variables, so its constructor will include two assignment statements shown in this implementation of one of the constructors:

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

implementing the constructor

The if-statement at the start checks the constructor's precondition. The first assignment statement, `manyItems = 0`, simply sets `manyItems` to zero, indicating that the bag does not yet have any elements. The second assignment statement, `data = new int[initialCapacity]`, is more interesting. This statement allocates an array of the right capacity (`initialCapacity`), and makes `data` refer to the new array. For example, suppose that `initialCapacity` is 6. After the two assignment statements, the instance variables look like this:

The private instance variables of this bag include an array of six integers. The bag does not yet contain any elements, so none of the array is being used.



Later, the program could add many elements to this bag, maybe even more than six. If there are more than six elements, then the bag's methods will increase the array's capacity as needed.

The other constructor is similar, except it always provides an initial capacity of ten.

The `add` Method. The `add` method checks that there is room to add a new element. If not, then the array capacity is increased before proceeding. (The new capacity is twice the old capacity plus 1. The extra +1 deals with the case where

the original size was zero.) The attempt to increase the array capacity may lead to an `OutOfMemoryError` or an arithmetic overflow as discussed on page 106. But usually these errors do not occur, and we can place the new element in the next available location of the array. What is the index of the next available location? For example, if `manyItems` is 3, then `data[0]`, `data[1]`, and `data[2]` are already occupied, and the next location is `data[3]`. In general, the next available location will be `data[manyItems]`. We can place the new element in `data[manyItems]`, as shown in this implementation:

implementing add

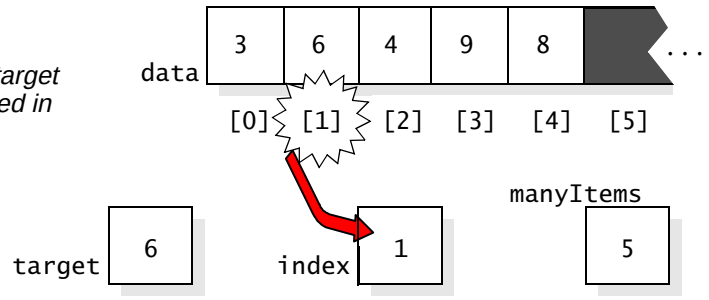
```
public void add(int element)
{
   if (manyItems == data.length)
   {
      // Double the capacity and add 1; this works even if manyItems is 0.
      // However, in the case that manyItems*2 + 1 is beyond
      // Integer.MAX_VALUE, there will be an arithmetic overflow and
      // the bag will fail.
      ensureCapacity(manyItems*2 + 1);
   }

   data[manyItems] = element;
   manyItems++;
}
```

See Self-Test Exercise 11 for an alternative approach to these steps.

Within a method we can activate other methods, such as the way that the `add` implementation activates `ensureCapacity` to increase the capacity of the array.

The `remove` Method. The `remove` method takes several steps to remove an element named `target` from a bag. In the first step, we find the index of `target` in the bag's array, and store this index in a local variable named `index`. For example, suppose that `target` is the number 6 in the bag drawn here:

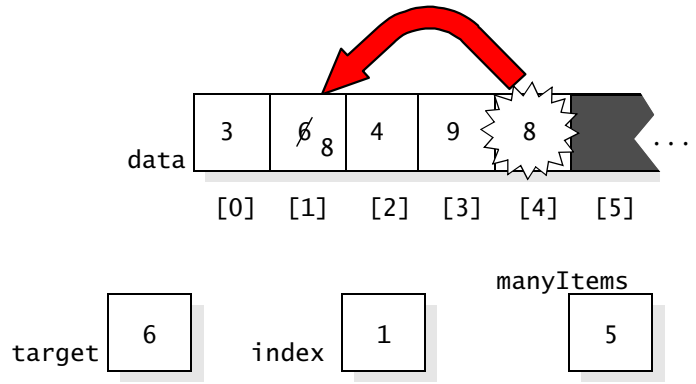
The index of the target is found and placed in a local variable called index.



In this example, `target` is a parameter to the `remove` method, `index` is a local variable in the `remove` method, and `manyItems` is the bag instance variable. As you can see in the drawing, the first step of `remove` is to locate the target (6) and place the index of the target in the local variable called `index`.

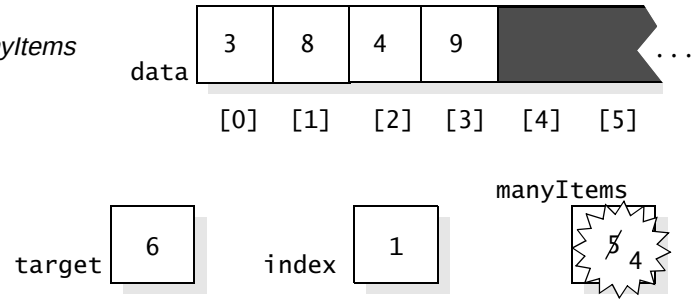
Once the index of the target is found, the second step is to take the *final* element in the bag and copy it to `data[index]`. The reason for this copying is so that all the bag's elements stay together at the front of the partially filled array, with no "holes." In our example, the number 8 is copied to `data[index]` as shown here:

The final element is copied onto the element that we are removing.



The third step is to reduce `manyItems` by one—in effect reducing the used part of the array by one. In our example, `manyItems` is reduced from 5 to 4:

The value of manyItems is reduced by one to indicate that one element has been removed.



The code for the `remove` method, shown in Figure 3.3, follows these three steps. There is also a check that the target is actually in the bag. If we discover that the target is not in the bag, then we do not need to remove anything. Also note that our method works correctly for the boundary values of removing the first or last element in the array.

implementing remove

Before we continue, we want to point out some programming techniques. Look at the following for-loop from Figure 3.3:

```
for (index = 0; (index < manyItems) && (target != data[index]); index++)
   // No work is needed in the body of this for-loop.
   ;
```

Instead of the usual loop body, there is merely a semicolon, which means that the body of this loop has no statements; all of the work is accomplished by the loop's three clauses. The first clause initializes `index` to zero. The second

clause indicates that the loop continues as long as `index` is still a location in the used part of the array (i.e., `index < manyItems`) and we have not yet found the target (i.e., `target != data[index]`). Each time through the loop, the third clause increments `index` by one (`index++`). No other work is needed in the loop, so the body of the loop has no statements.

A second programming technique concerns the boolean expression used to control the loop:

```
for (index = 0; (index < manyItems) && (target != data[index]); index++)
   // No work is needed in the body of this for-loop.
  ;
```

Look at the expression `data[index]` in the second part of the test. The valid indexes for `data` range from `0` to `manyItems-1`. But, if the target is not in the array, then `index` will eventually reach `manyItems`, which could be an invalid index. At that point, with `index` equal to `manyItems`, we must not evaluate the expression `data[index]`. Trying to evaluate `data[index]` with an invalid `index` will cause an `ArrayIndexOutOfBoundsException`.

The general rule: *Never use an invalid index with an array.*

FIGURE 3.3 Implementation of the Bag's Method to Remove an Element

Implementation

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

   // First, set index to the location of target in the data array,
   // which could be as small as 0 or as large as manyItems-1.
   // If target is not in the array, then index will be set equal to manyItems.
   for (index = 0; (index < manyItems) && (target != data[index]); index++)
      // No work is needed in the body of this for-loop.
      ;

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

See Self-Test Exercise 11 for an alternative approach to this step.

short-circuit evaluation of boolean expressions

Avoiding the invalid index is the reason for the first part of the boolean test (i.e., `index < manyItems`). Moreover, the test for `(index < manyItems)` must appear *before* the other part of the test. Placing `(index < manyItems)` first ensures that only valid indexes are used. The insurance comes from a technique called *short-circuit evaluation,* which Java uses to evaluate boolean expressions. In **short-circuit evaluation** a boolean expression is evaluated from left to right, and the evaluation stops as soon as there is enough information to determine the value of the expression. In our example, if `index` equals `manyItems`, then the first part of the boolean expression `(index < manyItems)` is `false`, so the entire `&&` expression *must* be `false`. It doesn't matter whether the second part of the `&&` expression is `true` or `false`. Therefore, Java doesn't bother to evaluate the second part of the expression, and the potential error of an invalid index is avoided.

The `countOccurrences` Method. To count the number of occurrences of a particular element in a bag, we step through the used portion of the partially filled array. Remember that we are using locations `data[0]` through `data[manyItems-1]`, so the correct loop is shown in this implementation:

implementing the countOccurrences method

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

   answer = 0;
   for (index = 0; index < manyItems; index++)
      if (target == data[index])
          answer++;
   return answer;
}
```

The `addAll` Method. The `addAll` method has this heading:

implementing the addAll method

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

The bag that activates `addAll` is increased by adding all the elements from `addend`. Our implementation follows these steps:

1. Ensure that the capacity of the bag is large enough to contain its current elements plus the extra elements that will come from `addend`, as shown here:

   ```
   ensureCapacity(manyItems + addend.manyItems);
   ```

   By the way, what happens in this statement if `addend` is null? Of course, a null value violates the precondition of `addAll`, but a programmer could mistakenly provide null. In that case, a `NullPointerException` will be thrown, and this possibility is documented in the specification of `addAll` on page 108.

2. Copy the elements from `addend.data` to the next available positions in our own `data` array. In other words, we will copy `addend.manyItems` elements from the front of `addend.data`. These elements go into our own data array beginning at the next available spot, `data[manyItems]`. We could write a loop to copy these elements, but a quicker approach is to use Java's `System.arraycopy` method, which has these five arguments:

```
System.arraycopy(source, si, destination, di, n);
```

the arraycopy method

The arguments `source` and `destination` are two arrays, and the other arguments are integers. The method copies `n` elements from `source` (starting at `source[si]`) to the `destination` array (with the elements being placed at `destination[di]` through `destination[di+n-1]`). For our purposes, we call the `arraycopy` method as shown here:

```
System.arraycopy
   (addend.data, 0, data, manyItems, addend.manyItems);
```

3. Increase our own `manyItems` by `addend.manyItems`, as shown here:

```
manyItems += addend.manyItems;
```

These three steps are shown in the `addAll` implementation of Figure 3.4.

FIGURE 3.4 Implementation of the Bag's `addAll` Method

Implementation

```
public void addAll(IntArrayBag addend)
{
   // If addend is null, then a NullPointerException is thrown.
   // In the case that the total number of items is beyond Integer.MAX_VALUE, there will be
   // arithmetic overflow and the bag will fail.
   ensureCapacity(manyItems + addend.manyItems);

   System.arraycopy(addend.data, 0, data, manyItems, addend.manyItems);
   manyItems += addend.manyItems;
}
```

implementing the union method

The `union` Method. The `union` method is different from our other methods. It is a *static* method, which means that it is not activated by any one bag object. Instead, the method must take its two parameters (bags `b1` and `b2`), combine these two bags together into a third bag, and return this third bag. The third bag is declared as a local variable called `answer` in the implementation of Figure 3.5. The capacity of the answer bag must be the sum of the capacities of `b1` and `b2`, so the actual answer bag is allocated by the statement:

```
answer = new IntArrayBag(b1.getCapacity( ) + b2.getCapacity( ));
```

This calls the `IntArrayBag` constructor to create a new bag with an initial capacity of `b1.getCapacity( ) + b2.getCapacity( )`.

The `union` implementation also makes use of the `System.arraycopy` method to copy elements from `b1.data` and `b2.data` into `answer.data`.

FIGURE 3.5 Implementation of the Bag's `union` Method

Implementation

```
public static IntArrayBag union(IntArrayBag b1, IntArrayBag b2)
{
   // If either b1 or b2 is null, then a NullPointerException is thrown.
   // In the case that the total number of items is beyond Integer.MAX_VALUE,
   // there will be an arithmetic overflow and the bag will fail.
   IntArrayBag answer =
      new IntArrayBag(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;
}
```

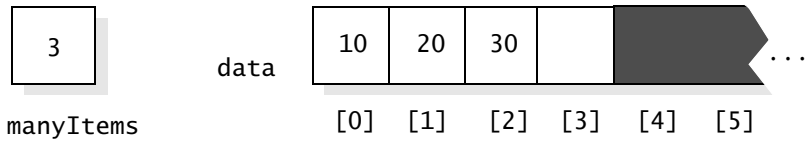
The `clone` Method. The `clone` method of a class allows a programmer to make a copy of an object. For example, the `IntArrayBag` class has a `clone` method to allow a programmer to make a copy of an existing bag. The copy is separate from the original, so that subsequent changes to the copy won't change the original, nor will subsequent changes to the original change the copy.

The `IntArrayBag` `clone` method will follow the pattern introduced in Chapter 2 on page 78. Therefore, the start of the `clone` method is:

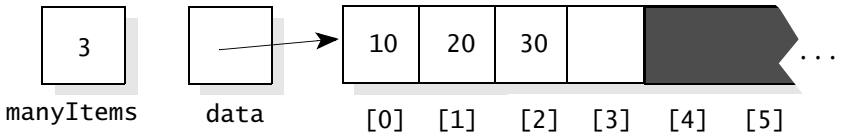
```
public Object clone( )
{ // Clone an IntArrayBag object.
   IntArrayBag answer;

   try
   {
      answer = (IntArrayBag) super.clone( );
   }
   catch (CloneNotSupportedException e)
   {
      throw new RuntimeException
      ("This class does not implement Cloneable.");
   }
   ...
```

As explained in Chapter 2, this code uses the `super.clone` method to make `answer` be an exact copy of the bag that activated the `clone` method. But for the bag class, an exact copy is not quite correct. The problem occurs because `super.clone` copies each instance variable of the class without concern for whether the instance variable is a primitive type (such as an `int`) or a more complicated type (such as an array or some other kind of reference to an object).

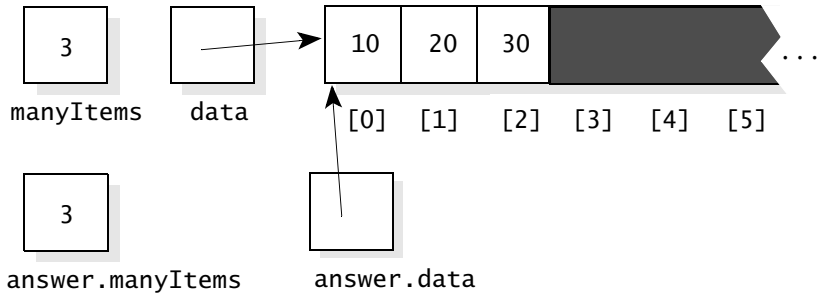
To see why this causes a problem, suppose we have a bag that contains three elements, as shown here:



This drawing uses the "array shorthand" that we've been using—just putting the name of the array right next to it. But in fact, as with every array, the instance variable `data` is actually a reference to the array, so a more accurate picture looks like the drawing at the top of the next page.



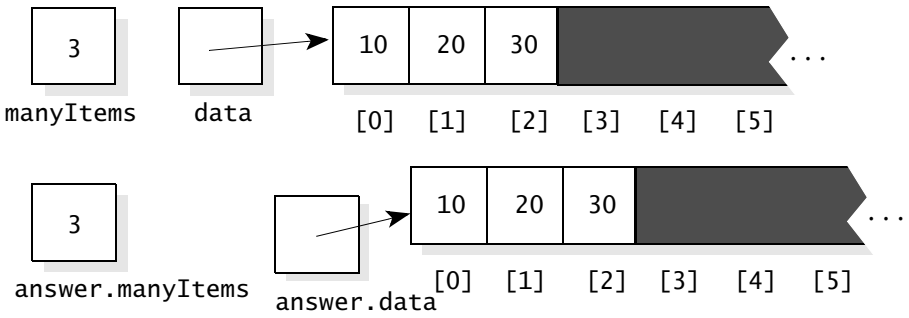
Now, suppose we activate `clone( )` to create a copy of this bag. The clone method executes the statement `answer = (IntArrayBag) super.clone( )`. What does `super.clone( )` do? It creates a new `IntArrayBag` object and `answer` will refer to this new `IntArrayBag`. But the new `IntArrayBag` has instance variables (`answer.manyItems` and `answer.data`) that are merely copied from the original. So, after the statement `answer = (IntArrayBag) super.clone( )` the situation looks like this (where `manyItems` and `data` are the instance variables from the original bag that activated the `clone` method):



As you can see, `answer.manyItems` has a copy of the number 3, and that is fine. But `answer.data` merely refers to the original's array. Subsequent changes to `answer.data` will affect the original and vice versa. This is incorrect behavior for a clone. To fix the problem, we need an additional statement before the return of the `clone` method. The purpose of the statement is to create a new array for the clone's data instance variable to refer to. Here's the statement:

```
answer.data = (int [ ]) data.clone( );
```

After this statement, `answer.data` refers to a separate array, as shown here:



The new `answer.data` array was created by creating a clone of the original array (as described on page 99). Subsequent changes to `answer` will not affect the original, nor will changes to the original affect `answer`. The complete `clone` method, including the extra statement at the end, is shown in Figure 3.6.

Programming Tip: Cloning a Class That Contains an Array

If a class has an instance variable that is an array, then the clone method needs extra work before it returns. The extra work creates a new array for the clone's instance variable to refer to.

The class may have other instance variables that are references to objects. In such a case, the clone method also carries out extra work. The extra work creates a new object for each such instances variable to refer to.

FIGURE 3.6 Implementation of the Bag's `clone` Method

Implementation

```
public Object clone( )
{
   { // Clone an IntArrayBag object.
   IntArrayBag answer;

   try
   {
      answer = (IntArrayBag) 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 = (int [ ]) data.clone( );

   return answer;
}
```

This step creates a new array for answer.data to refer to. The new array is separate from the original array so that subsequent changes to one will not affect the other.

FIGURE 3.7 Implementation of the Bag's `ensureCapacity` Method

Implementation

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

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

The `ensureCapacity` Method. This method ensures that a bag's array has at least a certain minimum length. Here is the method's heading:

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

The method checks whether the bag's array has a length below `minimumCapacity`. If so, then the method allocates a new larger array with a length of `minimumCapacity`. The elements are copied into the larger array, and the `data` instance variable is then made to refer to the larger array. Figure 3.7 shows our implementation, which follows the steps we have outlined.

The Bag ADT—Putting the Pieces Together

Three bag methods remain to be implemented: `size` (which returns the number of elements currently in the bag), `getCapacity` (which returns the current length of the bag's array, including the part that's not currently being used), and `trimToSize` (which reduces the capacity of the bag's array to equal exactly the current number of elements in the bag).

The `size` and `getCapacity` methods are implemented in one line each, and `trimToSize` is similar to `ensureCapacity`, so we won't discuss these methods. But you should examine these methods in the complete implementation file of Figure 3.8 on page 126. Also notice that the `IntArrayBag` class is placed in a package called `edu.colorado.collections`. Throughout the rest of this book, we will add other collection classes to this package.

FIGURE 3.8 Implementation File for the `IntArrayBag` Class

Implementation

```
// File: IntArrayBag.java from the package edu.colorado.collections
// Complete documentation is in Figure 3.1 on page 107 or from the IntArrayBag link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.collections;

public class IntArrayBag implements Cloneable
{
   // Invariant of the IntArrayBag 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 int[ ] data;
   private int manyItems;

   public IntArrayBag( )
   {
      final int INITIAL_CAPACITY = 10;
      manyItems = 0;
      data = new int[INITIAL_CAPACITY];
   }

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

   public void add(int element)
   {
      if (manyItems == data.length)
      {
         // Double the capacity and add 1; this works even if manyItems is 0. However, in
         // case that manyItems*2 + 1 is beyond Integer.MAX_VALUE, there will be an
         // arithmetic overflow and the bag will fail.
         ensureCapacity(manyItems*2 + 1);
      }
      data[manyItems] = element;
      manyItems++;
   }
```

(continued)

(FIGURE 3.8 continued)

```
public void addAll(IntArrayBag addend)
{
   // If addend is null, then a NullPointerException is thrown.
   // In the case that the total number of items is beyond Integer.MAX_VALUE, there will
   // be an arithmetic overflow and the bag will fail.
   ensureCapacity(manyItems + addend.manyItems);

   System.arraycopy(addend.data, 0, data, manyItems, addend.manyItems);
   manyItems += addend.manyItems;
}

public Object clone( )
{  // Clone an IntArrayBag object.
   IntArrayBag answer;

   try
   {
      answer = (IntArrayBag) 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 = (int [ ]) data.clone( );

   return answer;
}

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

   answer = 0;
   for (index = 0; index < manyItems; index++)
      if (target == data[index])
         answer++;
   return answer;
}
```

(continued)

(FIGURE 3.8 continued)

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

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

public int getCapacity( )
{
   return data.length;
}

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

   // First, set index to the location of target in the data array,
   // which could be as small as 0 or as large as manyItems-1.
   // If target is not in the array, then index will be set equal to manyItems.
   for (index = 0; (index < manyItems) && (target != data[index]); index++)
      // No work is needed in the body of this for-loop.
      ;

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

public int size( )
{
   return manyItems;
}
```

See Self-Test Exercise 11 for an alternative approach to this step.

(continued)

(FIGURE 3.8 continued)

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

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


    public static IntArrayBag union(IntArrayBag b1, IntArrayBag b2)
    {
       // If either b1 or b2 is null, then a NullPointerException is thrown.
       // In the case that the total number of items is beyond Integer.MAX_VALUE, there will
       // be an arithmetic overflow and the bag will fail.
       IntArrayBag answer = new IntArrayBag(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;
    }

}
```

Programming Tip: Document the ADT Invariant in the Implementation File

TIP 

The invariant of an ADT describes the rules that dictate how the instance variables are used. This information is important to the programmer who implements the class. Therefore, you should write this information in the implementation file, just before the declarations of the private instance variables. For example, the invariant for the `IntArrayBag` class appears before the declarations of `manyItems` and `data` in the implementation file of Figure 3.8 on page 126.

This is the best place to document the ADT's invariant. In particular, do not write the invariant as part of the class's specification, because a programmer who uses the ADT does not need to know about private instance variables. But the programmer who implements the ADT does need to know about the invariant.

The Bag ADT—Testing

Thus far, we have focused on the design and implementation of new classes and their methods. But it's also important to continue practicing the other aspects of software development, particularly testing. Each of the bag's new methods must be tested. As shown in Chapter 1, it is important to concentrate the testing on boundary values. At this point, we will alert you to only one potential pitfall, leaving the complete testing to Programming Project 2 on page 168.

PITFALL

Pitfall: An Object Can Be an Argument to Its Own Method

A class can have a method with a parameter that is the same data type as the class itself. For example, one of the `IntArrayBag` methods, `addAll`, has a parameter that is an `IntArrayBag` itself, as shown in this heading:

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

An `IntArrayBag` can be created and activate its `addAll` method using itself as the argument. For example:

```
IntArrayBag b = new IntArrayBag( );
b.add(5);
b.add(2);
b.addAll(b);
```

b now contains a 5 and a 2.

Now b contains two 5s and two 2s.

The highlighted statement takes all the elements in b (the 5 and the 2) and adds them to what's already in b, so b ends up with two copies of each number.

In the highlighted statement, the bag b is activating the `addAll` method, but this same bag b is the actual argument to the method. This is a situation that must be carefully tested. As an example of the danger, consider the incorrect implementation of `addAll` in Figure 3.9. Do you see what goes wrong with `b.addAll(b)`? (See the answer to Self-Test Exercise 12.)

FIGURE 3.9 Wrong Implementation of the Bag's `addAll` Method

A Wrong Implementation

```
public void addAll(IntArrayBag addend)
{
   int i; // An array index

   ensureCapacity(manyItems + addend.manyItems);
   for (i = 0; i < addend.manyItems; i++)
      add(addend.data[i]);
}
```

WARNING!

There is a bug in this implementation. See Self-Test Exercise 12.

The Bag ADT—Analysis

We finish this section with a time analysis of the bag's methods. Generally, we'll use the number of elements in a bag as the input size. For example, if `b` is a bag containing *n* integers, then the number of operations required by `b.countOccurrences` is a formula involving *n*. To determine the operations, we'll see how many statements are executed by the method, although we won't need an exact determination since our answer will use big-*O* notation. Except for two declarations and two statements, all of the work in `countOccurrences` happens in this loop:

```
for (index = 0; index < manyItems; index++)
   if (target == data[index])
      answer++;
```

We can see that the body of the loop will be executed exactly *n* times—once for each element in the bag. The body of the loop also has another important property: The body contains no other loops or calls to methods that contain loops. This is enough to conclude that the total number of statements executed by `countOccurrences` is no more than:

$n \times$ (number of statements in the loop) + 4

The extra +4 at the end is for the two declarations and two statements outside the loop. Regardless of how many statements are actually in the loop, the time expression is *always* $O(n)$—so the `countOccurrences` method is linear.

A similar analysis shows that `remove` is also linear, although `remove`'s loop sometimes executes fewer than *n* times. However, the fact that `remove` *sometimes* requires less than $n \times$ (number of statements in the loop) does not change the fact that the method is $O(n)$. In the worst case, the loop does execute a full *n* iterations, therefore the correct time analysis is no better than $O(n)$.

The analysis of the constructor is a special case. The constructor allocates an array of `initialCapacity` integers, and in Java all array components are initialized (integers are set to zero). The initialization time is proportional to the capacity of the array, so an accurate time analysis is $O($`initialCapacity`$)$.

constant time
O(1)

Several of the other bag methods do not contain any loops or array allocations. This is a pleasant situation because the time required for any of these methods does not depend on the number of elements in the bag. For example, when an element is added to a bag that does not need to grow, the new element is placed at the end of the array, and the `add` method never looks at the elements that were already in the bag. When the time required by a method does not depend on the size of the input, the procedure is called **constant time**, which is written $O(1)$.

The `add` method has two distinct cases. If the current capacity is adequate for a new element, then the time is $O(1)$. But if the capacity needs to be increased, then the time increases to $O(n)$ because of the array allocation and copying of elements from the old array to the new array.

The time analyses of all methods are summarized here for our `IntArrayBag`:

Operation	Time Analysis	
Constructor	$O(c)$	c is the initial capacity
add without capacity increase	$O(1)$	Constant time
add with capacity increase	$O(n)$	Linear time
b1.addAll(b2) without capacity increase	$O(n_2)$	Linear in the size of the added bag
b1.addAll(b2) with capacity increase	$O(n_1 + n_2)$	n_1 and n_2 are the sizes of the bags
clone	$O(c)$	c is the bag's capacity

Operation	Time Analysis	
countOccurrences	$O(n)$	linear time
ensure Capacity	$O(c)$	c is the specified minimum capacity
getCapacity	$O(1)$	Constant time
remove	$O(n)$	Linear time
size	$O(1)$	Constant time
trimToSize	$O(n)$	Linear time
union of b1 and b2	$O(c_1 + c_2)$	c_1 and c_2 are the bags' capacities

Self-Test Exercises

7. Draw a picture of `mybag.data` after these statements:
```
IntArrayBag mybag = new IntArrayBag(10);
mybag.add(1);
mybag.add(2);
mybag.add(3);
mybag.remove(1);
```
8. The bag in the previous question has a capacity of 10. What happens if you try to add more than ten elements to the bag?
9. Write the invariant of the bag ADT.
10. What is the meaning of a *static* method? How is the activation different than an ordinary method?
11. Use the expression `--manyItems` (with the `--` *before* `manyItems`) to rewrite the last two statements of `remove` (Figure 3.3 on page 118) as a single statement. If you are unsure of the difference between `manyItems--` and `--manyItems`, then go ahead and peek at our answer at the back of the chapter. Use `manyItems++` to make a similar alteration to the `add` method.
12. Suppose we implement `addAll` as shown in Figure 3.9 on page 130. What goes wrong with `b.addAll(b)`?

13. Describe the extra work that must be done at the end of the `clone` method. Draw pictures to show what goes wrong if this step is omitted.
14. Suppose `x` and `y` are arrays with 100 elements each. Use the `arraycopy` method to copy `x[10]...x[25]` to `y[33]...y[48]`.

3.3 PROGRAMMING PROJECT: THE SEQUENCE ADT

You are ready to tackle a collection class implementation on your own. The data type is called a **sequence**. A sequence is similar to a bag—both contain a bunch of elements. But unlike a bag, the elements in a sequence are arranged one after another.

how a sequence differs from a bag

How does this differ from a bag? After all, aren't the bag elements arranged one after another in the partially filled array that implements the bag? Yes, but that's a quirk of our particular bag implementation, and the order is just happenstance. Moreover, there is no way that a program using the bag can refer to the bag elements by their position in the array.

In contrast, the elements of a sequence are kept one after another, and the sequence's methods allow a program to step through the sequence one element at a time, using the order in which the elements are stored. Methods also permit a program to control precisely where elements are inserted and removed within the sequence.

The Sequence ADT—Specification

Our bag happened to be a bag of *integers*. We could have had a different underlying element type such as a bag of *double* numbers or a bag of *characters*. In fact, in Chapter 5, we'll see how to construct a collection that can simultaneously handle many different types of elements, rather than being restricted to one type of element. But for now, our collection classes will have just one kind of element for each collection. In particular, for our sequence class each element will be a *double* number, and the class itself is called `DoubleArraySeq`. We could have chosen some other type for the elements, but double numbers are as good as anything for your first implementation of a collection class.

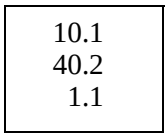
As with the bag, each sequence will have a current capacity, which is the number of elements the sequence can hold without having to request more memory. The initial capacity will be set by the constructor. The capacity can be increased in several different manners, which we'll see as we specify the various methods of the new class.

Constructor. The `DoubleArraySeq` has two constructors—a constructor that constructs an empty sequence with an initial capacity of 10, and another constructor that constructs an empty sequence with some specified initial capacity.

The `size` Method. The `size` method returns the number of elements in the sequence. Here is the heading:

```
public int size( )
```

For example, if `scores` is a sequence containing the values 10.1, 40.2, and 1.1, then `scores.size( )` returns 3. Throughout our examples, we will draw sequences vertically, with the first element on top, as shown in the picture in the margin (where the first element is 10.1).



Methods to Examine a Sequence. We will have methods to build a sequence, but it will be easier to explain first the methods to examine a sequence that has already been built. The elements of a sequence can be examined one after another, but the examination must be in order, from the first to the last. Three methods work together to enforce the in-order retrieval rule. The methods' headings are:

```
public void start( )
public double getCurrent( )
public void advance( )
```

When we want to retrieve the elements of a sequence, we begin by activating `start`. After activating `start`, the `getCurrent` method returns the first element of the sequence. Each time we call `advance`, the `getCurrent` method changes so that it returns the next element of the sequence. For example, if a sequence called `numbers` contains the four numbers 37, 10, 83, and 42, then we can write the following code to print the first three numbers of the sequence:

start, getCurrent, advance

```
numbers.start( );
System.out.println(numbers.getCurrent( ));
numbers.advance( );
System.out.println(numbers.getCurrent( ));
numbers.advance( );
System.out.println(numbers.getCurrent( ));
```

Prints 37

Prints 10

Prints 83

isCurrent

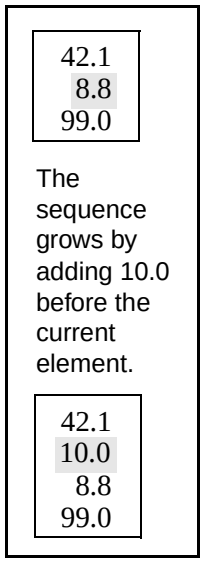
One other method cooperates with `getCurrent`. The `isCurrent` method returns a boolean value to indicate whether there actually is a current element for `getCurrent` to provide, or whether we have advanced right off the end of the sequence.

Using all four of the methods with a for-loop, we can print an entire sequence, as shown here for the `numbers` sequence:

```
for (numbers.start( ); numbers.isCurrent( ); numbers.advance( ))
   System.out.println(numbers.getCurrent( ));
```

The `addBefore` and `addAfter` Methods. There are two methods to add a new element to a sequence, with these headers:

```
public void addBefore(double element)
public void addAfter(double element)
```



The first method, `addBefore`, places a new element before the current element. For example, suppose that we have created the sequence shown in the margin, and that the current element is 8.8. In this example, we want to add 10.0 to our sequence, immediately before the current element. When 10.0 is added before the current element, other elements in the sequence—such as 8.8 and 99.0—will move down the sequence to make room for the new element. After the addition, the sequence has the four elements shown in the lower box.

If there is no current element, then `addBefore` places the new element at the front of the sequence. In any case, after the `addBefore` method returns, the new element will be the current element. In the example shown in the margin, the 10.0 becomes the new current element.

A second method, called `addAfter`, also adds a new element to a sequence—but the new element is added *after* the current element. If there is no current element, then the `addAfter` method places the new element at the end of the sequence (rather than the front). In all cases, when the method finishes, the new element will be the current element.

Either `addBefore` or `addAfter` can be used on an empty sequence to add the first element.

The `removeCurrent` Method. The current element can be removed from a sequence. The method for a removal has no parameters, but the precondition requires that there is a current element; it is this current element that is removed, as specified here:

- **removeCurrent**

  `public boolean removeCurrent( )`

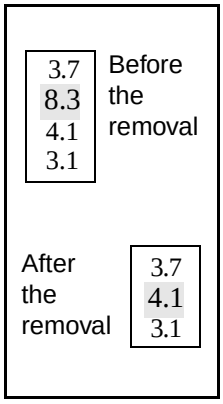
  Removes the current element from this sequence.

  Precondition:

  `isCurrent( )` returns `true`.

  Postcondition:

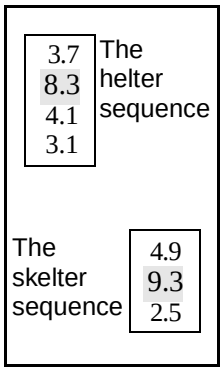
  The current element has been removed from this sequence. If this was the final element of the sequence (with nothing after it), then after the removal there is no longer a current element; otherwise the new current element is the one that used to be after the removed element.



For example, suppose `scores` is the four-element sequence shown at the top of the box in the margin, and the highlighted 8.3 is the current element. After activating `scores.removeCurrent( )`, the 8.3 has been deleted, and the 4.1 is now the current element.

The `addAll` Method. The `addAll` method is similar to the bag's `addAll` method. It allows us to place the contents of one sequence at the end of what we already have. The method has this heading:

```
public void addAll(DoubleArraySeq addend)
```



As an example, suppose we create two sequences called `helter` and `skelter`. The sequences contain the elements shown in the box in the margin (`helter` has four elements and `skelter` has three). We can then activate the method:

```
helter.addAll(skelter);
```

After the `addAll` activation, the `helter` sequence will have seven elements: 3.7, 8.3, 4.1, 3.1, 4.9, 9.3, 2.5 (its original four elements followed by the three elements of `skelter`). The current element of the `helter` sequence remains where it was (at the number 8.3), and the `skelter` sequence still has its original three elements.

The `concatenation` Method. The **concatenation** of two sequences is a new sequence obtained by placing one sequence after the other. We will implement `concatenation` with a static method that has the following two parameters:

```
public static DoubleArraySeq concatenation
(DoubleArraySeq s1, DoubleArraySeq s2)
```

A concatenation is somewhat similar to the union of two bags. For example:

```
DoubleArraySeq part1 = new DoubleArraySeq( );
DoubleArraySeq part2 = new DoubleArraySeq( );

part1.addAfter(3.7);
part1.addAfter(9.5);
part2.addAfter(4.0);
part2.addAfter(8.6);

DoubleArraySeq total = DoubleArraySeq.concatenation(part1, part2);
```

This computes the concatenation of the two sequences, putting the result in a third sequence.

After these statements, `total` is the sequence consisting of 3.7, 9.5, 4.0, 8.6. The new sequence computed by `concatenation` has no current element. The original sequences, `part1` and `part2`, are unchanged.

Notice the effect of having a *static* method: `concatenation` is not activated by any one sequence. Instead, the activation of

```
DoubleArraySeq.concatenation(part1, part2)
```

creates and returns a new sequence that includes the contents of `part1` followed by the contents of `part2`.

The `clone` Method. As part of our specification, we require that a sequence can be copied with a `clone` method. The clone contains the same elements as the original. If the original had a current element, then the clone has a current element in the corresponding place. For example:

```
DoubleArraySeq s = new DoubleArraySeq( );
s.addAfter(4.2);
s.addAfter(1.5);
s.start( );
IntArrayBag t = (DoubleArraySeq) s.clone( );
```

At the point when the clone is made, the sequence `s` has two elements (4.2 and 1.5) and the current element is the 4.2. Therefore, `t` will end up with the same two elements (4.2 and 1.5) and its current element will be the number 4.2. Subsequent changes to `s` will not affect `t`, nor vice versa.

Three Methods That Deal with Capacity. The sequence class has three methods for dealing with capacity—the same three methods that the bag has:

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

As with the bag, the purpose of these methods is to allow a programmer to explicitly set the capacity of the collection. If a programmer does not explicitly set the capacity, then the class will still work correctly, but some operations will be less efficient because the capacity might be repeatedly increased.

The Sequence ADT—Documentation

The complete specification for this first version of our sequence class is shown in Figure 3.10 on page 138. This specification is also available from the `DoubleArraySeq` link at the web address

```
http://www.cs.colorado.edu/~main/docs/
```

When you read the specification, you'll see that the package name is `edu.colorado.collections`. So, you should create a subdirectory called `edu/colorado/collections` for your implementation.

The specification also indicates some limitations—the same limitations that we saw for the bag class. For example, an `OutOfMemoryError` can occur in any method that increases the capacity. Several of the methods throw an `IllegalStateException` to indicate that they have been illegally activated (with no current element). Also, an attempt to move the capacity beyond the maximum integer causes the class to fail by an arithmetic overflow.

After you've looked through the specifications, we'll suggest a design that uses three private instance variables.

FIGURE 3.10 Specification for the `DoubleArraySeq` Class

Class DoubleArraySeq

❖ **public class DoubleArraySeq from the package edu.colorado.collections**

A `DoubleArraySeq` keeps track of a sequence of double numbers. The sequence can have a special "current element," which is specified and accessed through four methods that are not available in the bag class (`start`, `getCurrent`, `advance` and `isCurrent`).

Limitations:

(1) The capacity of a sequence can change after it's created, but the maximum capacity is limited by the amount of free memory on the machine. The constructor, `addAfter`, `addBefore`, `clone`, and `concatenation` will result in an `OutOfMemoryError` when free memory is exhausted.

(2) A sequence'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.

Specification

◆ **Constructor for the DoubleArraySeq**

```
public DoubleArraySeq(int initialCapacity)
```

Initialize an empty sequence with a specified initial capacity. Note that the `addAfter` and `addBefore` methods work efficiently (without needing more memory) until this capacity is reached.

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

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

◆ **Second Constructor for the DoubleArraySeq**

```
public DoubleArraySeq(int initialCapacity)
```

Initialize an empty sequence with a specified initial capacity. Note that the `addAfter` and `addBefore` methods work efficiently (without needing more memory) until this capacity is reached.

Parameters:
`initialCapacity` – the initial capacity of this sequence

Precondition:
`initialCapacity` is non-negative.

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

Throws: `IllegalArgumentException`
Indicates that `initialCapacity` is negative.

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

(continued)

(FIGURE 3.10 continued)

- **addAfter and addBefore**

```
public void addAfter(double element)
public void addBefore(double element)
```

Adds a new element to this sequence, either before or after the current element. If this new element would take this sequence 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 sequence. If there was a current element, then `addAfter` places the new element after the current element and `addBefore` places the new element before the current element. If there was no current element, then `addAfter` places the new element at the end of this sequence and `addBefore` places the new element at the front of this sequence. In all cases, the new element becomes the new current element of this sequence.

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

Note:
An attempt to increase the capacity beyond `Integer.MAX_VALUE` will cause this sequence to fail with an arithmetic overflow.

- **addAll**

```
public void addAll(DoubleArraySeq addend)
```

Place the contents of another sequence at the end of this sequence.

Parameters:
`addend` – a sequence whose contents will be placed at the end of this sequence

Precondition:
The parameter, `addend`, is not null.

Postcondition:
The elements from `addend` have been placed at the end of this sequence. The current element of this sequence remains where it was, and the `addend` is also unchanged.

Throws: `NullPointerException`
Indicates that `addend` is null.

Throws: `OutOfMemoryError`
Indicates insufficient memory to increase the capacity of this sequence.

Note:
An attempt to increase the capacity beyond `Integer.MAX_VALUE` will cause this sequence to fail with an arithmetic overflow.

(continued)

(FIGURE 3.10 continued)

- **advance**

  `public void advance( )`

  Move forward, so that the current element is now the next element in this sequence.

  Precondition:
  `isCurrent( )` returns `true`.

  Postcondition:
  If the current element was already the end element of this sequence (with nothing after it), then there is no longer any current element. Otherwise, the new element is the element immediately after the original current element.

  Throws: `IllegalStateException`
  Indicates that there is no current element, so `advance` may not be called.

- **clone**

  `public Object clone( )`

  Generate a copy of this sequence.

  Returns:
  The return value is a copy of this sequence. Subsequent changes to the copy will not affect the original, nor vice versa. The return value must be typecast to a `DoubleArraySeq` before it is used.

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

- **concatenation**

  `public static DoubleArraySeq concatenation (DoubleArraySeq s1, DoubleArraySeq s2)`

  Create a new sequence that contains all the elements from one sequence followed by another.

  Parameters:
  `s1` – the first of two sequences
  `s2` – the second of two sequences

  Precondition:
  Neither `s1` nor `s2` is null.

  Returns:
  a new sequence that has the elements of `s1` followed by the elements of `s2` (with no current element)

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

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

  Note:
  An attempt to increase the capacity beyond `Integer.MAX_VALUE` will cause this sequence to fail with an arithmetic overflow.

(continued)

(FIGURE 3.10 continued)

- **ensureCapacity**

  `public void ensureCapacity(int minimumCapacity)`

  Change the current capacity of this sequence.

  Parameters:

  `minimumCapacity` – the new capacity for this sequence

  Postcondition:

  This sequence's capacity has been changed to at least `minimumCapacity`.

  Throws: `OutOfMemoryError`

  Indicates insufficient memory for: `new double[minimumCapacity]`.

- **getCapacity**

  `public int getCapacity( )`

  Accessor method to determine the current capacity of this sequence. The `addBefore` and `addAfter` methods works efficiently (without needing more memory) until this capacity is reached.

  Returns:

  the current capacity of this sequence

- **getCurrent**

  `public double getCurrent( )`

  Accessor method to determine the current element of this sequence.

  Precondition:

  `isCurrent( )` returns `true`.

  Returns:

  the current element of this sequence

  Throws: `IllegalStateException`

  Indicates that there is no current element.

- **isCurrent**

  `public boolean isCurrent( )`

  Accessor method to determine whether this sequence has a specified current element that can be retrieved with the `getCurrent` method.

  Returns:

  `true` (there is a current element) or `false` (there is no current element at the moment)

- **removeCurrent**

  `public void removeCurrent( )`

  Remove the current element from this sequence.

  Precondition:

  `isCurrent( )` returns `true`.

  Postcondition:

  The current element has been removed from this sequence, and the following element (if there is one) is now the new current element. If there was no following element, then there is now no current element.

  Throws: `IllegalStateException`

  Indicates that there is no current element, so `removeCurrent` may not be called. *(continued)*

(FIGURE 3.10 continued)

- **size**

  `public int size( )`

  Accessor method to determine the number of elements in this sequence.

  Returns:
  the number of elements in this sequence

- **start**

  `public void start( )`

  Set the current element at the front of this sequence.

  Postcondition:
  The front element of this sequence is now the current element (but if this sequence has no elements at all, then there is no current element).

- **trimToSize**

  `public void trimToSize( )`

  Reduce the current capacity of this sequence to its actual size (i.e., the number of elements it contains).

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

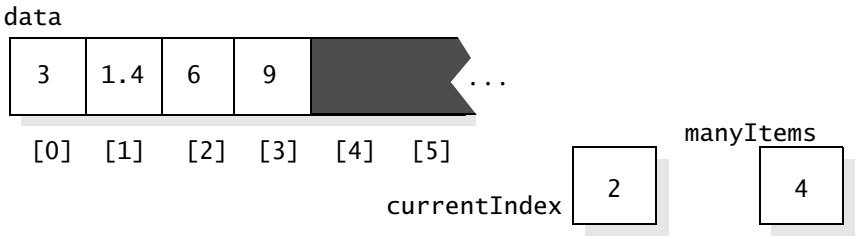
  Throws: `OutOfMemoryError`
  Indicates insufficient memory for altering the capacity.

The Sequence ADT—Design

Our suggested design for the sequence ADT has three private instance variables. The first variable, `data`, is an array that stores the elements of the sequence. Just like the bag, `data` is a partially filled array, and a second instance variable, called `manyItems`, keeps track of how much of the `data` array is currently being used. Therefore, the used part of the array extends from `data[0]` to `data[manyItems-1]`. The third instance variable, `currentIndex`, gives the index of the current element in the array (if there is one). Sometimes a sequence has no current element, in which case `currentIndex` will be set to the same number as `manyItems` (since this is larger than any valid index). The complete invariant of our ADT is stated as three rules:

1. The number of elements in the sequence is stored in the instance variable `manyItems`.
2. For an empty sequence (with no elements), we do not care what is stored in any of `data`; for a nonempty sequence, the elements of the sequence are stored from the front to the end in `data[0]` to `data[manyItems-1]`, and we don't care what is stored in the rest of `data`.
3. If there is a current element, then it lies in `data[currentIndex]`; if there is no current element, then `currentIndex` equals `manyItems`.

As an example, suppose that a sequence contains four numbers, with the current element at `data[2]`. The instance variables of the object might appear as shown here:



In this example, the current element is at `data[2]`, so the `getCurrent( )` method would return the number 6. At this point, if we called `advance( )`, then `currentIndex` would increase to 3, and `getCurrent( )` would then return the 9.

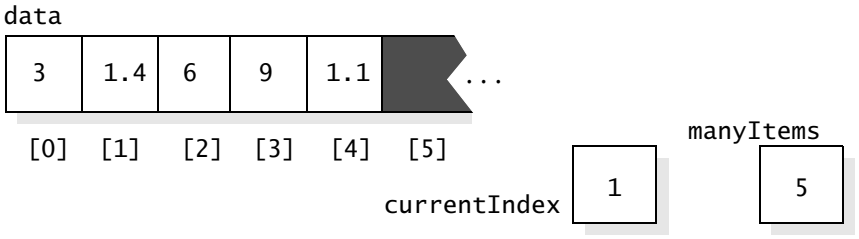
Normally, a sequence has a current element, and the instance variable `currentIndex` contains the location of that current element. But if there is no current element, then `currentIndex` contains the same value as `manyItems`. In the above example, if `currentIndex` was 4, then that would indicate that there is no current element. Notice that this value (4) is beyond the used part of the array (which stretches from `data[0]` to `data[3]`).

invariant of the ADT

The stated requirements for the instance variables form the invariant of the sequence ADT. You should place this invariant at the top of your implementation file (`DoubleArraySeq.java`). We will leave most of this implementation file up to you, but we will offer some hints and a bit of pseudocode.

The Sequence ADT—Pseudocode for the Implementation

The `removeCurrent` Method. This method removes the current element from the sequence. First check that the precondition is valid (use `isCurrent( )`). Then remove the current element by shifting each of the subsequent elements leftward one position. For example, suppose we are removing the current element from the sequence drawn here:

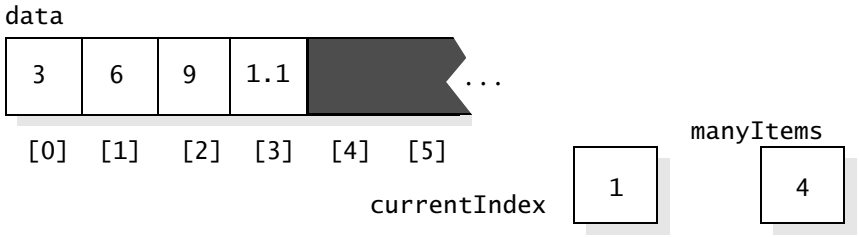


What is the current element in this picture? It is the 1.4, since `currentIndex` is 1 and `data[1]` contains 1.4.

In the case of the bag, we could remove an element such as 1.4 by copying the final element (1.1) onto the 1.4. But this approach won't work for the *sequence* because the elements would lose their sequence order. Instead, each element after the 1.4 must be moved leftward one position. The 6 moves from `data[2]` to `data[1]`; the 9 moves from `data[3]` to `data[2]`; the 1.1 moves from `data[4]` to `data[3]`. This is a lot of movement, but a small for-loop suffices to carry out all the work. This is the pseudocode:

```
for (i = the index after the current element; i < manyItems; i++)
    Move an element from data[i] back to data[i-1];
```

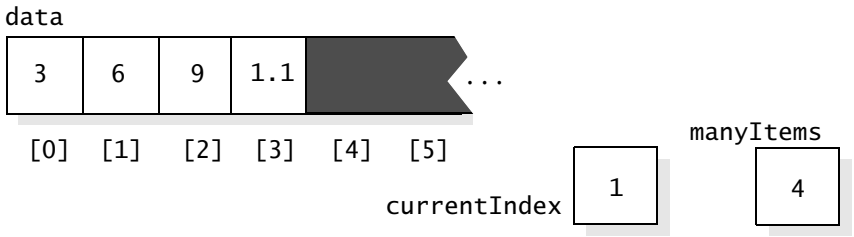
When the loop completes, you should reduce `manyItems` by one. The final result for our example is:



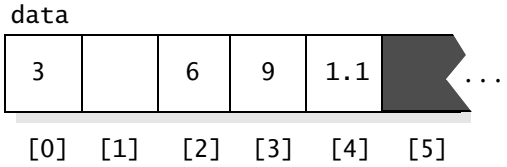
After the removal, the value in `currentIndex` is unchanged. In effect, this means that the element that was just after the removed element is now the current element. You must check that the method works correctly for boundary values—removing the first element and removing the end element. In fact, both these cases work fine. When the end element is removed, `currentIndex` will end up with the same value as `manyItems`, indicating that there is no longer a current element.

The `addBefore` Method. If there is a current element, then `addBefore` must take care to put the new element just before the current position. Elements that are already at or after the current position must be shifted rightward to make room for the new element. We suggest that you start by shifting elements at the end of the array rightward one position each until you reach the position for the new element.

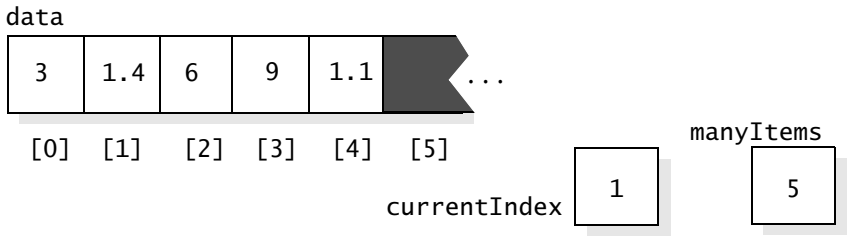
For example, suppose you are putting 1.4 at the location `data[1]` in the following sequence:



You would begin by shifting the 1.1 rightward from `data[3]` to `data[4]`; then move the 9 from `data[2]` to `data[3]`; then the 6 moves from `data[1]` rightward to `data[2]`. At this point, the array looks like this:



Of course, `data[1]` actually still contains a 6 since we just copied the 6 from `data[1]` to `data[2]`. But we have drawn `data[1]` as an empty box to indicate that `data[1]` is now available to hold the new element (the 1.4 that we are puting in the sequence). At this point we can place the 1.4 in `data[1]` and add one to `manyItems`, as shown here:



The pseudocode for shifting the elements rightward uses a for-loop. Each iteration of the loop shifts one element, as shown here:

```
for (i = manyItems; data[i] is the wrong spot for element ; i--)
   data[i] = data[i-1];
```

The key to the loop is the test `data[i] is the wrong spot for element`. How do we test whether a position is the wrong spot for the new element? A position is wrong if `(i > currentIndex)`. Can you now write the entire method in Java? (See the solution to Self-Test Exercise 15, and don't forget to handle the special case when there is no current element.)

Other Methods. The other sequence methods are straightforward; for example, the `addAfter` method is similar to `addBefore`. Some additional useful methods are described in Programming Project 4 on page 168. You'll also need to be careful that you don't mindlessly copy the implementation of a bag method. For example, the `concatenation` method is similar to the bag's `union` method, but there is one extra step that concatenation must take (it sets `currentIndex` to `manyItems`).

Self-Test Exercises

15. Write the sequence's `addBefore` method.
16. Suppose that a sequence has 24 elements, and there is no current element. According to the invariant of the ADT, what is `currentIndex`?
17. Suppose `g` is a sequence with 10 elements and you activate `g.start( )` and then activate `g.advance( )` three times. What value is then in `g.currentIndex`?
18. What are good boundary values to test the `removeCurrent` method?
19. Write a demonstration program that asks the user for a list of family member ages, then prints the list in the same order that it was given.
20. Write a new method to remove a specified element from a sequence. The method has one parameter (the element to remove).
21. For a sequence of numbers, suppose that you insert 1, then 2, then 3, and so on up to *n*. What is the big-*O* time analysis for the combined time of inserting all *n* numbers with `addAfter`? How does the analysis change if you insert *n* first, then *n*–1, and so on down to 1—always using `addBefore` instead of `addAfter`?
22. Which of the ADTs—the bag or the sequence—*must* be implemented by storing the elements in an array? (Hint: We are not beyond asking a trick question.)

3.4 APPLETS FOR INTERACTIVE TESTING

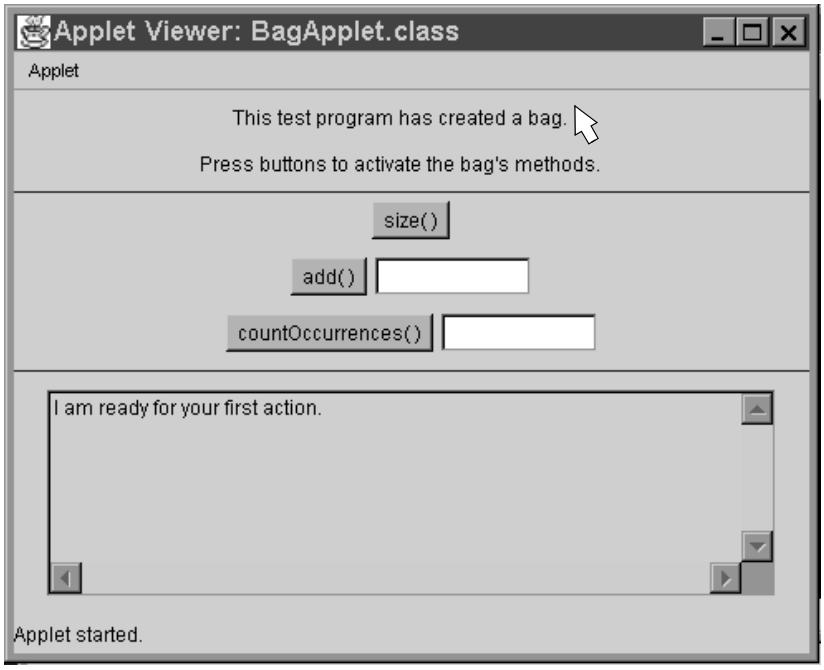
When you implement a new class, it's useful to have a small interactive test program to help you test the class methods. Such a program can be written as a Java **applet**, which is a Java program written in a special format to have a graphical user interface. The graphical user interface is also called a GUI (pronounced "gooey"), and it allows a user to interact with a program by clicking the mouse, typing information into boxes, and performing other familiar actions. With a Java applet, GUIs are easy to create even if you've never run into such goo before.

This section shows a pattern for developing such applets. To illustrate the pattern, we'll implement an applet that lets you test three of the bag's methods (`size`, `add`, and `countOccurrences`). When the bag applet starts, a GUI is created, similar to the drawing in Figure 3.11(a).

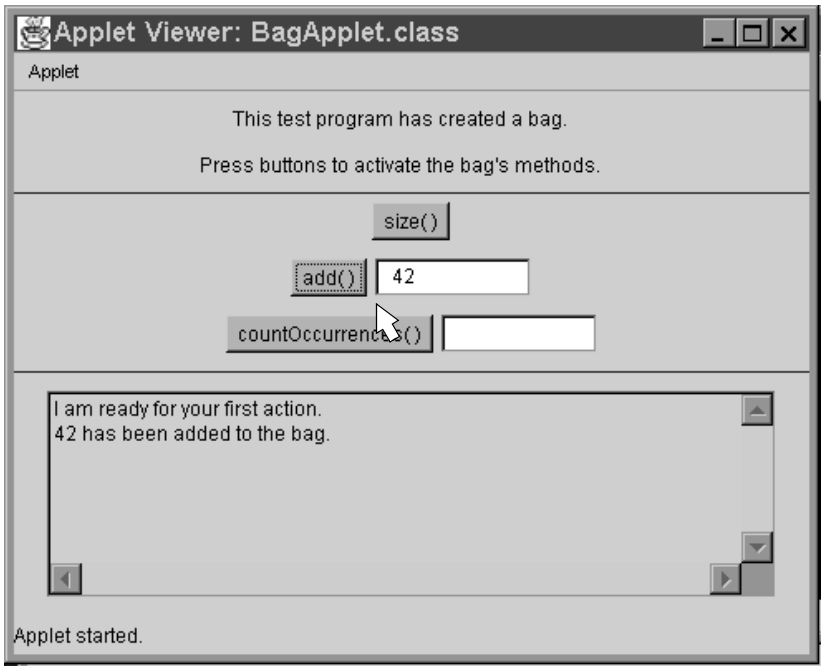
By the way, the word "applet" means a particular kind of Java program, so you might show Figure 3.11 to your boss and say, "My applet created this nice GUI." But you can also use the word "applet" to talk about the GUI itself, such as "The applet in Figure 3.11(a) has three buttons in its middle section." And in fact, there are three buttons in that applet—the rectangles labeled `size( )`, `add( )`, and `countOccurrences( )`.

FIGURE 3.11 Two Views of the Applet to Test the `IntArrayBag` Class

(a) When the applet first opens, the applet has the components shown here.



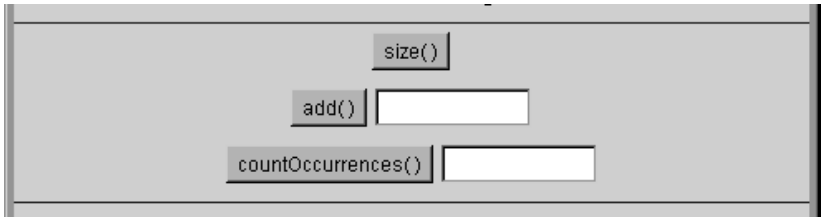
(b) The user interacts with the applet by typing information and clicking on the buttons with the mouse. In this example, the user has typed 42 into the add text field, and then clicked the add button. The applet responds with a message "42 has been added to the bag," written in the text area at the bottom of the applet.



The applet in Figure 3.11 is intended to be used by the programmer who wrote the `IntArrayBag` class, to check interactively that the class is working correctly. When the applet starts, two sentences appear at the top: "`This test program has created a bag. Press buttons to activate the bag's methods.`" Above these sentences are some extra items, shown here:

The display above our sentences is created automatically by the applet display mechanism. The exact form of this display varies from one system to another, but the dark bar across the top generally contains controls such as the ✕ in the top right corner. Clicking on that ✕ with the mouse closes the applet on this particular system.

A series of buttons appears in the middle part of the applet, like this:



To test the bag, the user clicks on the various buttons. For example, the user can click on `size()` and a new message will appear in the large text area at the bottom of the applet. The message will tell the current size of the bag, as obtained by activating the `size()` method. If you click on this button right at the start, you'll get the message "`The bag's size is 0.`"

The user can also activate `add` or `countOccurrences`, but these methods each need an argument. For example, to add the number 42 to the bag, the user types the number 42 in the white box next to the `add` button, then clicks `add()`. The result of adding 42 is shown in Figure 3.11(b). After elements have been added, the user can test `countOccurrences`. For example, to count the occurrences of the number 10, the user types 10 in the box by the `countOccurrences` button and then clicks `countOccurrences()`. The applet activates `countOccurrences(10)` and prints the method's return value in the large text area at the bottom.

Anyway, that's the behavior that we want. Let's look at an outline of the Java programming techniques to produce such behavior, as shown in Figure 3.12.

FIGURE 3.12 Outline for the Interactive Applet to Test the `IntArrayBag` class

Java Applet Outline

```
// FILE: BagApplet.java
```

1. Import statements. These statements import the class that is being tested and also the Java classes that are needed by the applet. For this applet, we must import the `IntArrayBag` class:

```
import edu.colorado.collections.IntArrayBag;
```

Most applets will also have these three import statements:

```
import java.applet.Applet; // Provides the Applet class.
import java.awt.*;         // Provides Button class, etc.
import java.awt.event.*;   // Provides ActionEvent, ActionListener.
```

2. The class definition.

```
public class BagApplet extends Applet
{
    // Declare an IntArrayBag object for the Applet to manipulate:
    IntArrayBag b = new IntArrayBag( );
```

3. Declarations of the applet's components. These are the declarations of buttons, text areas, and other GUI components that appear in the applet.

```
    public void init( )
    {
        ...
    }
```

4. The `init` method.

5. Implementations of the action listeners. This code tells the applet what to do when each of the buttons is pressed.

6. Implementations of other methods. These are methods that are activated from within `init` to carry out various subtasks.

```
}
```

Six Parts of a Simple Interactive Applet

Figure 3.12 on page 149 shows an outline for the Java code of the applet that tests the `IntArrayBag`. The same outline can be used for an applet that interactively tests any class. The code has six parts, which we'll discuss now.

1. Import statements. As with any Java program, we begin with a collection of import statements to tell the compiler about the other classes that we'll be using. In the case of the bag applet, we import the `IntArrayBag` class (using the statement `import edu.colorado.collections.IntArrayBag;`). Most applets also have these three import statements:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
```

abstract windowing toolkit

The first import statement provides a class called `Applet`, which we'll use in a moment. The other two import statements provide items from the **abstract windowing toolkit** (the "AWT"), which is a collection of classes for drawing buttons and other GUI items.

2. The class definition. After the import statements, we define a class, much like any other Java class. This class definition begins with the line:

```
public class IntArrayBag Applet extends Applet
```

inheritance: the BagApplet gets a bunch of methods from the Applet class

The definition continues down to the last closing bracket of the file. The class for the bag applet is called `BagApplet`, which is certainly a good name, but what does "`extends Applet`" mean? It means that the `BagApplet` class will not be written entirely by us. Instead, the class begins by already having all the nonprivate methods of another class called `Applet`. We imported the `Applet` class from `java.applet.Applet`, and it is provided as part of the Java language so that a class such as `BagApplet` does not have to start from scratch. The act of obtaining methods from another class is called **inheritance**. The class that provides these methods (such as the `Applet` class) is called the **superclass**, and the new class (such as `BagApplet`) is called the **extended class**. Chapter 13 studies inheritance in detail, but for now all you need to know is that the `BagApplet` obtains a bunch of methods from the `Applet` class without having to do anything more than "`extends Applet`."

At the top of the class we define an `IntArrayBag` instance variable:

```
IntArrayBag b = new IntArrayBag( );
```

This bag, `b`, will be manipulated when the user clicks on the applet's buttons. In general, an interactive test applet will have one or more objects declared here, and these objects are manipulated by clicking the applet's buttons.

3. Declarations of the applet's components. An applet's components are the buttons and other items that are displayed when the applet runs. These components are declared as instance variables of the class. Our bag applet has several kinds of components: buttons (such as size()), text fields (which are the white rectangles next to some of the buttons), and a text area (which is the large rectangle in the bottom third of the applet). In all, there are six important components in the bag applet, represented by these six instance variables:

```
Button    sizeButton             = new Button("size( )");
Button    addButton              = new Button("add( )");
TextField elementText            = new TextField(10);
Button    countOccurrencesButton = new Button("countOccurrences( )");
TextField targetText             = new TextField(10);
TextArea  feedback               = new TextArea(7, 60);
```

All the instance variables are declared near the top of the class definition, before any of the method definitions. They cannot have the usual private access because they'll be accessed from other classes that we'll see shortly. But before that, let's look at the three kinds of components: button, text field, and text area.

button

A **button** is a grey rectangle with a label. When a button is created, the constructor is given a string that is printed in the middle of the button. For example, this declaration creates a button called `sizeButton`, and the label on the button is the string "`size( )`":

```
Button sizeButton = new Button("size( )");
```

The bag applet has three `Button` objects: `sizeButton`, `addButton`, and `countOccurrencesButton`.

text field

A **text field** is a white rectangle that can display one line of text. A text field is set up so that the program's user can click on the field and type information, and the applet can then read that information. Our applet has two text fields, one next to the `add` button and one next to the `countOccurrences` button. The `TextField` class has a constructor with one argument—an integer that specifies approximately how many characters can fit in the text field. For example, one of our text fields is declared as:

```
TextField elementText = new TextField(10);
```

The `elementText` text field can hold about 10 characters. The user can actually type beyond 10 characters, but only 10 characters of a long string will be displayed. We plan to display `elementText` right beside the `add` button, like this:

To test the `add` method, the user will type a number in the text field and click on the `add` button.

text area

A **text area** is like a text field with more than one line. Its constructor has two arguments that specify the number of rows and columns of text to display. Our bag applet has one text area, declared like this:

```
TextArea feedback = new TextArea(7, 60);
```

This large text area appears at the bottom of the applet. The intention is to use the text area to display messages to the user.

The declarations we have seen created the three kinds of components: `Button`, `TextField`, and `TextArea`. All three classes are part of the `java.awt` package that is imported by our applet. When we declare a button (or other component) and create it with the constructor, it does not immediately appear in the GUI. How do the objects get placed in the GUI? Also, how does the applet know what to do when the user clicks on a button or takes some other action? The answers to these two questions lie in a special applet method called `init`, which we'll discuss next.

4. The `init` method. A Java application program has a special static method called `main`. A Java applet does not have `main`. Instead, an applet has a special nonstatic method called `init`. When an applet runs, the runtime system creates an object of the applet class, and activates `init()` for that object. There are several other applet methods that the runtime system also activates at various times, but an interactive test program needs only `init`.

Our `init` method carries out four kinds of actions:

A. The add method. We can add one of the interactive components to the GUI. This is done with an applet method called `add`. The method has one argument, which is the component that is being added to the GUI. For example, one of our buttons is `sizeButton`, so we can write the statement:

the applet's add method

```
add(sizeButton);
```

As components are added, the GUI fills up from left to right. If there is no room for a component on the current line, then the GUI moves down and starts a new row of components. Later you can learn more sophisticated ways of laying out the components of a GUI, but the simple left-to-right method used by an applet is a good starting point.

B. Displaying messages. We can display messages in the GUI. Each message is a fixed string that provides some information to the user. Each of these messages is a `Label` object (from the package `java.awt`). To create and display a message, we activate `add`, with a newly created `Label` as the argument. For example:

```
add(new Label("This test program has created a bag"));
```

printing a message in the GUI

The `Label` constructor has one argument, which is the string that you want to display. The add method will put the message in the next available spot of the GUI.

C. New lines and horizontal lines. If our applet class has other methods (besides `init`), then we can activate these other methods. For example, we plan to have two other methods in the `IntArrayBag` class:

addNewLine
addHorizontalLine

```
void addNewLine( );
void addHorizontalLine(Color c);
```

The `addNewLine` method forces the GUI to start a new line, even if there's room for more components on the current line. The second method, `addHorizontalLine`, draws a horizontal line in the specified color. We'll have to define these two methods as part of `BagApplet.Java`, but they won't be difficult. (The data type `Color` is part of `java.lang`. It includes `Color.blue` and twelve other colors plus the ability to define your own colors.)

D. Activate methods of the components. The buttons and other components have methods that can be activated. For example, one of the methods of a `TextArea` is called `append`. The method has one argument, which is a string, and this string is appended to the end of what's already in the text field. One of the statements in our `init` method will activate `append` in this way:

append

```
feedback.append("I am ready for your first action.\n");
```

This causes the message "`I am ready for your first action.`" to be written in the `feedback` text field (with a newline character `\n` at the end of the message).

action listener objects

The most important method for buttons involves a new kind of object called an action listener. An action listener is object that an applet programmer creates to describe the action that should be taken when certain events occur. Our bag applet will have a different kind of action listener for each of the three buttons:

Kind of Action Listener	Purpose
`SizeListener`	Describes the actions to be taken when `sizeButton` is clicked.
`AddListener`	Describes the actions to be taken when `addButton` is clicked.
`CountOccurrencesListener`	Describes the actions to be taken when `countOccurrencesButton` is clicked.

Each kind of action listener is actually a new class that we'll define in a moment. But the only thing you need to know for the `init` method is how to connect an action listener to a `Button`. The solution is to activate a

method called `addActionListener` for each `Button`. For example, to connect `sizeButton` to its action listener, we place this statement in the `init` method:

```
sizeButton.addActionListener(new SizeListener( ));
```

Notice that `addActionListener` is a method of the `Button` class, and its one argument is a new `SizeListener` object. Of course, we still need to implement the `SizeListener` class, as well as the other two action listener classes. But first, let's summarize all the pieces that are part of the `init` method for the `BagApplet`. Within `init`, we expect to activate these methods to carry our work:

- `add`—an `Applet` method to add the buttons and other components to the display
- `addNewLine` and `addHorizontalLine`—two methods that we will write for the `BagApplet`
- `feedback.append`—a method of the `TextField` to place the message "I am ready for your first action" in `feedback` (a `TextField` object)
- `addActionListener`—a method that will be called once for each of the three buttons

The complete `init` implementation is shown in Figure 3.13 on page 156. We've used just one method that we haven't yet mentioned. That one method (`setEditable`) is summarized in Figure 3.14 on page 157 along with the other applet-oriented methods that we have used or plan to use.

5. Implementations of the action listeners. The next step of the applet implementation is to design and implement three action listener classes—one for each of our three buttons. The purpose of an action listener is to describe the actions that are carried out when a button is pushed.

Here's the Java syntax for defining an action listener class—the blank line is filled in with your choice of a name for the action listener class.

```
class _____________ implements ActionListener
{
   void actionPerformed(ActionEvent event)
   {
      ...
   }
}
```

The phrase "`implements ActionListener`" informs the Java compiler that the class will have a certain method that is specified in the `ActionListener` interface that is part of `java.awt.*`. The method, called `actionPerformed`, is shown with "..." to indicate its body. The `actionPerformed` method will be executed when an action occurs in the action listener's component, such as

clicking a button. For example, here is the complete definition of the action listener that handles the clicking of the size() button of our test applet:

an action listener for sizeButton

```
class SizeListener implements ActionListener
{
   void actionPerformed(ActionEvent event)
   {
      feedback.append("The bag has size " + b.size( ) + ".\n");
   }
}
```

This declares a class called `SizeListener`, which includes its own `actionPerformed` method. For most classes, the class definition would go in a separate file called `SizeListener.java`. But a separate file is undesirable here because the `actionPerformed` method needs access to two instance variables: the bag `b` and the text area `feedback`. The necessary access can be provided by placing the entire `SizeListener` definition within the `BagApplet`. This is an example of an **inner class,** where the definition of one class is placed inside of another. An inner class has two key properties:

- The larger class that encloses an inner class may use the inner class; but the inner class may not be used elsewhere.
- The inner class may access nonprivate instance variables and methods of the larger class. Some Java implementations also permit an inner class to access private instance variables of the larger class. But other implementations forbid private access from an inner class. (Java implementations that are built into web browsers are particularly apt to forbid the private access.)

So, by making `SizeListener` an inner class, the `actionPerformed` method can activate `feedback.append` to print a message in the `feedback` component of the applet. The message itself includes an activation of `b.size( )`, so an entire message is something like "`The bag has size 42.`"

The actionPerformed Method

The `SizeListener` class is an inner class, declared within `BagApplet`. Therefore, its `actionPerformed` method has access to the instance variables of the `BagApplet`.

By the way, the `actionPerformed` method has a parameter called `event`. For more complex actions, the `event` can provide more information about exactly which kind of action triggered the `actionPerformed` method.

(Text continues on page 158)

FIGURE 3.13 Implementation of the BagApplet's `init` Method

Implementation

```
public void init( )
{
    // Some messages for the top of the Applet:
    add(new Label("This test program has created a bag."));
    add(new Label("Press buttons to activate the bag's methods."));
    addHorizontalLine(Color.blue);

    // The Button for testing the size method:
    add(sizeButton);
    addNewLine( );

    // The Button and TextField for testing the add method:
    add(addButton);
    add(elementText);
    addNewLine( );

    // The Button and TextField for testing the countOccurrences method:
    add(countOccurrencesButton);
    add(targetText);
    addNewLine( );

    // A TextArea at the bottom to write messages:
    addHorizontalLine(Color.blue);
    addNewLine( );
    feedback.setEditable(false);
    feedback.append("I am ready for your first action.\n");
    add(feedback);

    // Tell the Buttons what they should do when they are clicked:
    sizeButton.addActionListener(new SizeListener( ));
    addButton.addActionListener(new AddListener( ));
    countOccurrencesButton.addActionListener(new CountOccurrencesListener( ));

}
```

FIGURE 3.14 Guide to Building an Applet for Interactive Testing

Methods to Call from an Applet or from a Class That Extends an Applet	
`add(component)`	*The component may be any of Java's AWT components such as Button, TextArea, or TextField. As components are added, the applet fills up from left to right. If there is no room for a component on the current line, then the applet moves down and starts a new row of components.*
`addNewLine( )` `addHorizontalLine(Color c)`	*These are not actually Applet methods—you'll need to define them if you want to use them (see page 160).*

Constructors for Three Useful Applet Components	
`Button(String label)`	*Creates a button with a given label.*
`TextField(int size)`	*Creates a white box for the user to type information. The size is the number of characters.*
`TextArea(int rows, int columns)`	*Creates a box with the given number of rows and columns—often for displaying information to the user.*

Six Useful Methods for a Component	
`b.setActionListener (ActionListener act)`	*We use b.setActionListener for a Button b. The ActionListener, act, describes the actions to take when the Button b is pressed. See page 154 for information on how to create an ActionListener.*
`t.append(String message)`	*We use t.append for a TextArea t. The specified message is added to the end of the TextArea.*
`t.getText( )`	*We use t.getText for a TextField t. The method returns a copy of the String that the user has typed in the field.*
`t.setEditable(boolean editable)`	*The component t can be a TextArea or a TextField. The boolean parameter tells whether you want the user to be able to type text into the component.*
`t.requestFocus( )` `t.selectAll( )`	*We use these methods with a TextField. The requestFocus method causes the mouse to go to the field, and selectAll causes all text to be highlighted.*
`c.setSize(int width, int height)`	*This method may be used with any component c. The component's width and height are set to the given values in pixels.*

registering an ActionListener

Once an action listener is created, it must be registered with its particular button. The registration is made in the `init` method. Our applet had these three statements to register the three `ActionListener` objects:

```
sizeButton.addActionListener(new SizeListener( ));
addButton.addActionListener(new AddListener( ));
countOccurrencesButton.addActionListener
   (new CountOccurrencesListener( ));
```

For example, the first of these statements creates a new `SizeListener` and registers it with the button `sizeButton`.

Let's look at the second action listener class for our applet: `AddListener`. This action listener handles the actions of `addButton`, which is shown here along with the `TextField` that's right beside it in the applet:

What actions should occur when the user clicks the `addButton`? The text should be read from the `TextField`. This text is a `String`, such as "42", but it can be converted to its value as an integer by using the Java method `Integer.parseInt`. The method `Integer.parseInt` has one argument (a `String` which should contain an integer value), and the return value is the `int` value of the `String`. Once we know the value of the integer provided by the user, we can add it to the bag `b` and print an appropriate message in the applet's feedback area. Following these ideas, we have this first try at implementing `AddListener`:

```
class AddListener implements ActionListener
{
   void actionPerformed(ActionEvent event)
   {
      String userInput = elementText.getText( );
      int element = Integer.parseInt(userInput);
      b.add(element);
      feedback.append(element + " has been added to the bag.\n");
   }
}
```

The `actionPerformed` method defined here uses three of the applet's instance variables: (1) `elementText`, which is the `TextField` where the user typed a number; (2) the bag `b`, where the new element is added; and (3) the `TextArea feedback`, where a message is printed providing feedback to the user.

The method works fine, though a problem arises if the user forgets to type a number in the `TextField` before clicking the button. In this case, a `NumberFormatException` will occur when `Integer.parseInt` tries to convert the user's string to an integer.

catching the possible exception

The best solution to this problem is to "catch" the exception when it occurs, rather than allowing the exception to stop the applet. The syntax for catching a `NumberFormatException` looks like this:

```
try
{
   ...code that might throw a NumberFormatException...
}
catch (NumberFormatException e)
{
   ...code to execute if the NumberFormatException happens...
}
```

The words `try` and `catch` are Java keywords for handling exceptions. The full power of `try` and `catch` are described in Appendix C. For our purposes, we'll follow the preceding pattern to write a better version of `AddListener`:

an action listener for addButton

```
class AddListener = implements ActionListener
{
   void actionPerformed(ActionEvent event)
   {
      try
      {
         String userInput = elementText.getText( );
         int element = Integer.parseInt(userInput);
         b.add(element);
         feedback.append
            (element + " has been added to the bag.\n");
      }
      catch (NumberFormatException e)
      {
         feedback.append
            ("Type an integer before clicking button.\n");
         elementText.requestFocus( );
         elementText.selectAll( );
      }
   }
}
```

If a `NumberFormatException` occurs, then the code in the `catch` block is executed. This code prints a message in the feedback area of the applet, then activates two methods for `elementText` (which is the `TextField` where the user was supposed to type a number):

```
elementText.requestFocus( );
elementText.selectAll( );
```

requestFocus and selectAll

The `requestFocus` method causes the mouse cursor to jump into the `TextField`, and the `selectAll` method causes any text in the field to be highlighted. So now, if the user forgets to type a number, the applet will print a nice error message and provide a second chance.

Our applet needs one more action listener for the `countOccurrences` button. That implementation is part of Figure 3.2 on page 112.

6. Implementations of other methods. Our applet has two other methods that we've mentioned: (1) `addHorizontalLine`, which draws a horizontal line in a specified color; and (2) `addNewLine`, which causes a new line to start in the GUI, even if there's room for more components on the current line.

Our `addHorizontalLine` doesn't really draw a line. Instead, it adds a component called a `Canvas` to the applet. A `Canvas` is another applet component, like a `Button`, primarily used for drawing graphical images. The size of the `Canvas` can be set in **pixels**, which are the individual dots on a computer screen. Today's typical screens have about 100 pixels per inch, so a `Canvas` that is only one pixel high looks like a horizontal line. Here's our implementation:

implementation of addHorizontalLine

```
private void addHorizontalLine(Color c)
{
   // Add a Canvas 10000 pixels wide but
   // only 1 pixel high, which acts as
   // a horizontal line.
   Canvas line = new Canvas( );
   line.setSize(10000, 1);
   line.setBackground(c);
   add(line);
}
```

Notice that the Canvas is 10,000 pixels wide, which is wide enough to span even the largest applet—at least on today's computer screens.

implementation of addNewLine

Our last method, `addNewLine`, works by calling `addHorizontalLine` with the color set to the background color of the applet. In effect, we are drawing a horizontal line, but it is invisible since it's the same color as the applet's background.

The implementation of `addNewLine` is given in Figure 3.15 as part of the complete applet. Look through the implementation with an eye toward how it can be expanded to test all of the bag's methods or to test a different class such as the `DoubleArraySeq` class.

FIGURE 3.15 Complete Implementation of the BagApplet

Java Applet Implementation

```
// File: BagApplet.java
// This applet is a small example to illustrate how to write an interactive applet that
// tests the methods of another class. This first version tests three of the IntArrayBag methods.

import edu.colorado.collections.IntArrayBag;
import java.applet.Applet;
import java.awt.*;          // Imports Button, Canvas, TextArea, TextField
import java.awt.event.*;    // Imports ActionEvent, ActionListener

public class BagApplet extends Applet
{
   // An IntArrayBag for this applet to manipulate:
   IntArrayBag b = new IntArrayBag( );

   // These are the interactive components that will appear in the applet.
   // We declare one Button for each IntArrayBag method that we want to be able to
   // test. If the method has an argument, then there is also a TextField
   // where the user can enter the value of the argument.
   // At the bottom, there is a TextArea to write messages.
   Button    sizeButton             = new Button("size( )");
   Button    addButton              = new Button("add( )");
   TextField elementText            = new TextField(10);
   Button    countOccurrencesButton = new Button("countOccurrences( )");
   TextField targetText             = new TextField(10);
   TextArea  feedback               = new TextArea(7, 60);

   public void init( )
   {
      || See the implementation in Figure 3.13 on page 156.
   }

   class SizeListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         feedback.append("The bag has size " + b.size( ) + ".\n");
      }
   }
```

(continued)

(FIGURE 3.15 continued)

```
class AddListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      try
      {
         String userInput = elementText.getText( );
         int element = Integer.parseInt(userInput);
         b.add(element);
         feedback.append(element + " has been added to the bag.\n");
      }
      catch (NumberFormatException e)
      {
         feedback.append("Type an integer before clicking button.\n”);
         elementText.requestFocus( );
         elementText.selectAll( );
      }
   }
}

class CountOccurrencesListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      try
      {
         String userInput = targetText.getText( );
         int target = Integer.parseInt(userInput);
         feedback.append(target + " occurs ");
         feedback.append(b.countOccurrences(target) + “times.\n”);
      }
      catch (NumberFormatException e)
      {
         feedback.append("Type a target before clicking button.\n");
         targetText.requestFocus( );
         targetText.selectAll( );
      }
   }
}
```

(continued)

(FIGURE 3.15 continued)

```
    private void addHorizontalLine(Color c)
    {
        // Add a Canvas 10000 pixels wide but only 1 pixel high, which acts as
        // a horizontal line to separate one group of components from the next.
        Canvas line = new Canvas( );
        line.setSize(10000,1);
        line.setBackground(c);
        add(line);
    }

    private void addNewLine( )
    {
        // Add a horizontal line in the background color. The line itself is
        // invisible, but it serves to force the next component onto a new line.
        addHorizontalLine(getBackground( ));
    }

}
```

How to Compile and Run an Applet

An applet can be compiled just like any other Java program. For example, using the Java Development Kit we can compile `BagApplet.java` with the command line:

```
javac BagApplet.java
```

You may have some other way of compiling Java programs in your development environment, but the result will be the same. The act of compiling produces the file `BagApplet.class`. The compilation will probably produce three other files with names such as `BagApplet$SizeListener.class`. These are the compiled versions of the inner classes.

applets were created to be viewed over the Internet

Applets were actually created to run as part of a page that you view over the Internet with a web browser. These pages are called **html pages**, which stands for "hyper-text markup language." So, to run the `BagApplet`, we need a small html file. The file, called `BagApplet.html`, should be created by you in the same directory as `BagApplet.class`, and it should contain the two lines of html code shown at the top of the next page.

```
<applet code="BagApplet.class" width=480 height=340>
</applet>
```

The first line, containing `<applet...>` tells the web browser that you are going to start an applet. Usually, you will have at least three pieces of information about the applet:

`code = "BagApplet.class"`	*Tells the browser where to find the compiled class.*
`width = 480` `height = 340`	*Sets the applet's size in pixels. Today's typical screens have about 100 pixels per inch, so a size of 480 x 340 is about 4.8 inches by 3.4 inches.*

Many Java development environments have a feature to automatically create a small html file such as this.

Once the html file is in place, you can run the applet in one of two ways. One approach is to run an **appletviewer,** which is a tool that reads an html file and runs any applets that it finds. The Java Development Kit has an appletviewer that is executed from the command line. For example, to run the JDK appletviewer you change to the directory that contains `BagApplet.html` and type the command:

```
appletviewer BagApplet.html
```

This command runs the applet, resulting in the display shown in Figure 3.11 on page 147.

The applet can also be displayed by putting it in a location that's available to your web browser. My latest information about this approach is available at `http://www.cs.colorado.edu/~main/java.html`.

Beyond the `init` Method

Our test applet needed to define only the `init` method. More complex applets can also be created, involving graphical images plus interaction. Graphical applets will generally provide other methods called `start`, `paint`, `update`, `stop`, and `destroy`. A good resource is *Graphic Java Mastering the AWT* by David M. Geary.

Self-Test Exercises

23. Write three declarations of instance variables that might appear in an applet. Include a constructor activation in each declaration. The first declaration is a button with the label "Mmm, good." The second declaration is for a text

field of 15 characters. The third declaration is for a text area with 12 rows and 60 columns.

24. Write three statements that could appear in the `init` method of an applet. The statements should take the three components from the previous exercise and add them to the applet's GUI.
25. Write a statement that could appear in the `init` method of an applet to display the message "FREE Consultation!"
26. Describe the technique used in the implementation of `addHorizontalLine`.
27. Write a new action listener that can be registered to a button of the `BagApplet`. The `actionPerformed` method should print feedback to indicate how many copies of the numbers 1 through 10 appear in the applet's bag.
28. Suppose that `b` is a button in the `BagApplet`. Write a statement that could appear in the `init` method to create an action listener of the previous exercise and register it to the button `b`.
29. Suppose that `s` is a `String` that may or may not contain a sequence of digits representing a valid integer. Write a `try-catch` statement that will try to convert the `String` to an integer and print two times this integer. If the `String` does not represent a valid integer, then an error message should be printed.

CHAPTER SUMMARY

- A *collection class* is an ADT where each object contains a collection of elements. Bags and sequences are two examples of collection classes.
- The simplest implementations of collection classes use a *partially filled array*. Using a partially filled array requires each object to have at least two instance variables: the array itself and an `int` variable to keep track of how much of the array is being used.
- When a collection class is implemented with a partially filled array, the capacity of the array should grow as elements are added to the collection. The class should also provide explicit methods to allow a programmer to control the capacity of the array.
- In a collection class, some methods allocate additional memory (such as changing the capacity of an array). These methods have the possibility of throwing an `OutOfMemoryError` (when the machine runs out of memory).
- A class may have other instance variables that are references to objects or arrays. In such a case, the `clone` method must carry out extra work. The extra work creates a new object or array for each such instance variable to refer to.

- When you design an ADT, always make an explicit statement of the rules that dictate how the instance variables are used. These rules are called the *invariant of the ADT*, and should be written at the top of the implementation file for easy reference.
- Small ADTs can be tested effectively with an *interactive test applet* that follows the standard format of the `BagApplet` in Section 3.4.

? Solutions to Self-Test Exercises

1.
```
int i;
int[ ] b;
b = new int[1000];
for (i = 1; i <= 1000; i++)
   b[i-1] = i;
```

2. `b.length`

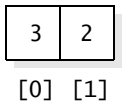
3. 42 (since `a` and `b` refer to the same array)

4. 0 (since `b` is a clone of `a`)

5. The array referred to by the parameter in the method is the same as the array referred to by the actual argument. So, the actual argument will have its first component changed to 42.

6.
```
void copyFront(int[] a, int[] b, int n)
// Precondition: a.length and b.length are
// both greater than or equal to n.
// Postcondition: n integers have been cop-
// ied from the front of a to the front of b.
{
   int i;
   for (i = 0; i < n; i++)
      b[i] = a[i];
}
```

7.



We don't care what appears beyond `data[1]`.

8. When the 11th element is added, the `add` method will increase the capacity.

9. See the two rules on page 114.

10. A static method is not activated by any one particular object. It is activated by writing the class name, a dot, the method name, and the argument list. For example:
```
IntArrayBag.union(b1, b2)
```

11. The two statements can be replaced by one: `data[index] = data[--manyItems];` When `--manyItems` appears as an expression, the variable `manyItems` is decremented by one, and the resulting value is the value of the expression. (On the other hand, if `manyItems--` appears as an expression, the value of the expression is the value of `manyItems` prior to subtracting one.) Similarly, the last two statements of `add` can be combined to `data[manyItems++] = element;`

12. For the incorrect implementation of `addAll`, suppose we have a bag `b` and we activate `b.addAll(b)`. Then the private instance variable `manyItems` is the same variable as `addend.manyItems`. Each iteration of the loop adds 1 to `manyItems`, and hence `addend.manyItems` is also increasing, and the loop never ends.

One warning: Some collection classes in the Java libraries have an `addAll` method that fails for the statement `b.addAll(b)`. The reason is improved efficiency. So, before you use an `addAll` method, check the specification for restrictions.

13. At the end of the clone implementation we need an additional statement to make a separate copy of the data array for the clone to use. If we don't make this copy, then our own data array and the clone's data array will be one and the same (see the pictures on page 123).

14. `System.arrayCopy(x, 10, y, 33, 16);`

15.
```
void addBefore(double element)
{
  int i;

  if (manyItems == data.length)
  {  // Try to double the capacity
     ensureCapacity(manyItems*2 + 1);
  }

  if (!isCurrent( ))
     currentIndex = 0;
  for
  (i=manyItems; i>currentIndex; i--)
     data[i] = data[i-1];
  data[currentIndex] = element;
  manyItems++;
}
```

16. 24

17. `g.currentIndex` will be 3 (since the 4th element occurs at `data[3]`).

18. The `removeCurrent` method should be tested when the sequence's size is just 1, and when the sequence is at its full capacity. At full capacity you should try removing the first element, and the last element of the sequence.

19. Your program can be similar to Figure 3.2 on page 111.

20. Here is our method's heading, with a postcondition:

```
void remove(int target);
// Postcondition: If target was in the
// sequence, then the first copy of target
// has been removed, and the element after
// the removed element (if there is one)
// becomes the new current element; other-
// wise the sequence remains unchanged.
```

The easiest implementation searches for the index of the target. If this index is found, then set `currentIndex` to this index, and activate the ordinary `removeCurrent` method.

21. The total time to add 1, 2, ... , n with `addAfter` is $O(n)$. The total time to add n, n–1, ..., 1 with `addBefore` is $O(n^2)$. The larger time for the second approach is because an addition at the front of the sequence requires all of the existing elements to be shifted right to make room for the new element. Hence, on the second addition, one element is shifted. On the third addition, two elements are shifted. And so on to the n^{th} element which needs n–1 shifts. The total number of shifts is 1+2+...+(n–1), which is $O(n^2)$. (To show that this sum is $O(n^2)$, use a technique similar to Figure 1.3 on page 21.)

22. Neither of the classes *must* use an array. In later chapters we will see both classes implemented without arrays.

23.
```
Button b = new Button("Mmm, good");
TextField f = new TextField(15);
TextArea a = new TextArea(12, 60);
```

24.
```
add(b); add(f); add(a);
```

25.
```
add
(new Label("FREE Consultation!"));
```

26. The "horizontal line" is actually a `Canvas` that is one pixel high and very wide.

27.
```
class InfoListener
implements ActionListener
{
  public void
  actionPerformed(ActionEvent event)
  {
    int i;
    int count;

    for (i = 1; i <= 10; i++)
    {
      count = b.countOccurrences(i);
      feedback.append
      (i + " occurs " + count + ".\n");
    }
  }
}
```

28.
```
b.addActionListener
(new InfoListener( ));
```

29.
```
try
{
   int value = Integer.parseInt(s);
   System.out.println(2 * value);
}
catch (NumberFormatException e)
{
   System.out.println("Not number");
}
```

PROGRAMMING PROJECTS

1 For the `IntArrayBag` class, implement a new method called `equals` with a boolean return value and one parameter. The parameter, called `b`, is another `IntArrayBag`. The method returns `true` if `b` and the bag that activates the method have exactly the same number of every element. Otherwise the method returns `false`. Notice that the locations of the elements in the data arrays are not necessarily the same. It is only the number of occurrences of each element that must be the same.

The worst-case time for the method should be $O(mn)$, where m is the size of the bag that activates the method and n is the size of `b`.

2 A **black box** test of a class is a program that tests the correctness of a class without directly examining the private instance variables of the class. You can imagine that the private instance variables are inside an opaque black box where they cannot be seen, so all testing must occur only through activating the public methods.

Write a noninteractive black box test program for the `IntArrayBag` class. Make sure that you test the boundary values, such as an empty bag, a bag with just one element, and a full bag.

3 Expand the `BagApplet` from Figure 3.15 on page 161. The expanded version should have three bags and buttons to activate any method of any bag. Also include a button that will carry out an action such as:

`b1 = IntArrayBag.union(b2, b3)`.

4 Implement the sequence class from Section 3.3. You may wish to provide some additional useful methods, such as:
(1) a method to add a new element at the front of the sequence; (2) a method to remove the element at the front of the sequence; (3) a method to add a new element at the end of the sequence; (4) a method that makes the last element of the sequence become the current element; (5) a method that returns the i^{th} element of the sequence (starting with the 0^{th} at the front); (6) a method that makes the i^{th} element become the current element.

5 Implement an applet for interactive testing of the sequence class from the previous project.

6 A bag can contain more than one copy of an element. For example, the chapter describes a bag that contains the number 4 and two copies of the number 8. This bag behavior is different from a **set**, which can contain only a single copy of any given element. Write a new collection class called `SetOfInt`, which is similar to a bag, except that a set can contain only one copy of any given element. You'll need to change the specification a bit. For example, instead of the bag's `countOccurrences` method, you'll want a method such as this:

```
boolean contains(int target)
// Postcondition: The return value is true if
// target is in the set; otherwise the return
// value is false.
```

Make an explicit statement of the invariant of the set ADT. Do a time analysis for each operation. At this point, an efficient implementation is not needed. For example, just adding a new element to a set will take linear time because you'll need to check that the new element isn't already present. Later we'll explore more efficient implementations.

You may also want to add additional methods to your set ADT, such as a method for subtracting one set from another.

7 Rewrite the sequence class using a new class name, `DoubleArraySortedSeq`. In the new class, the add method always puts the new element so that all the elements stay in order from smallest to largest. There is no `addBefore` or `addAfter` method. All the other methods are the same as the original sequence ADT.

8 A one-variable **polynomial** is an arithmetic expression of the form:

$$a_0 + a_1x + a_2x^2 + \ldots + a_kx^k$$

The highest exponent, k, is called the **degree** of the polynomial, and the constants a_0, a_1, … are the **coefficients.** For example, here are two polynomials with degree three:

$$2.1 + 4.8x + 0.1x^2 + (-7.1)x^3$$

$$2.9 + 0.8x + 10.1x^2 + 1.7x^3$$

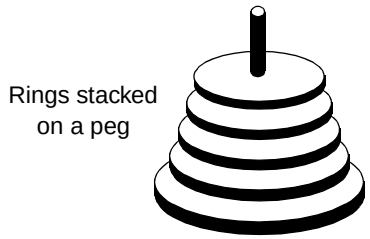
Specify, design, and implement a class for polynomials. Spend some time thinking about operations that make sense on polynomials. For example, you can write a method that adds two polynomials. Another method should evaluate the polynomial for a given value of x.

9 Specify, design, and implement a class that can be one player in a game of tic-tac-toe. The constructor should specify whether the object is to be the first player (X's) or the second player (O's). There should be a method to ask the object to make its next move, and a method that tells the object what the opponent's next move is. Also include other useful methods, such as a method to ask whether a given spot of the tic-tac-toe board is occupied, and if so, whether the occupation is with an X or an O. Also, include a method to determine when the game is over, and whether it was a draw, an X win, or an O win.

Use the class in two programs: a program that plays tic-tac-toe against the program's user, and a program that has two tic-tac-toe objects that play against each other.

10 Specify, design, and implement a collection class that can hold up to five playing cards. Call the class `PokerHand`, and include a method with a boolean return value to allow you to compare two poker hands. For two hands `x` and `y`, the relation `x.beats(y)` means that `x` is a better hand than `y`. If you do not play in a weekly poker game yourself, then you may need to consult a card rule book for the rules on the ranking of poker hands.

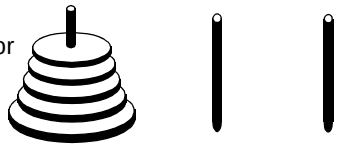
11 Specify, design, and implement a class that keeps track of rings stacked on a peg, rather like phonograph records on a spindle. An example with five rings is shown here:



The peg may hold up to 64 rings, with each ring having its own diameter. Also, there is a rule that requires each ring to be smaller than any ring underneath it. The class's methods should include: (a) a constructor that places n rings on the peg (where n may be as large as 64). These n rings have diameters from n inches (on the bottom) to one inch (on the top); (b) an accessor method that returns the number of rings on the peg; (c) an accessor method that returns the diameter of the topmost ring; (d) a method that adds a new ring to the top (with the diameter of the ring as a parameter to the method); (e) a method that removes the topmost ring; (f) a method that prints some clever representation of the peg and its rings. Make sure that all methods have appropriate preconditions to guarantee that the rule about ring sizes is enforced. Also spend time designing appropriate private instance variables.

12 In this project, you will design and implement a class called `Towers`, which is part of a program that lets a child play a game called Towers of Hanoi. The game consists of three pegs and a collection of rings that stack on the pegs. The rings are different sizes. The initial configuration for a five-ring game is shown here, with the first tower having rings from one inch (on the top) to five inches (on the bottom).

Initial configuration for a five-ring game of Towers of Hanoi

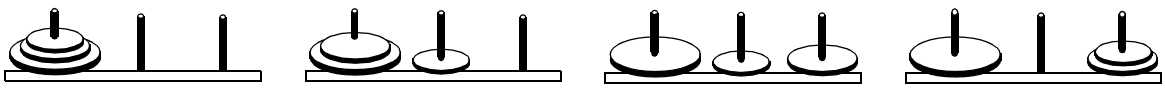


The rings are stacked in decreasing order of their size, and the second and third towers are initially empty. During the game, the child may transfer rings one-at-a-time from the top of one peg to the top of another. The goal is to move all the rings from the first peg to the second peg. The difficulty is that the child may not place a ring on top of one with a smaller diameter. There is the one extra peg to hold rings temporarily, but the prohibition against a larger ring on a smaller ring applies to it as well as the other two pegs. A solution for a three-ring game is shown at the bottom of the page. The `Towers` class must keep track of the status of all three pegs. You might use an array of three pegs, where each peg is an object from the previous project. The `Towers` methods are specified in the next column.

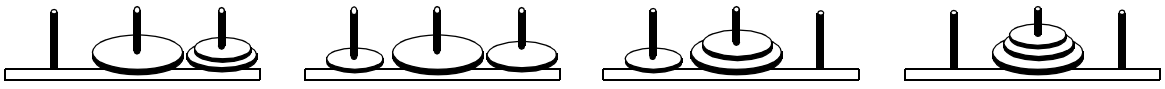
```
Towers(int n);
// Precondition: 1 <= n <= 64.
// Postcondition: The towers have been initialized
// with n rings on the first peg and no rings on
// the other two pegs. The diameters of the first
// peg's rings are from one inch (on the top) to n
// inches (on the bottom).

int countRings(int pegNumber)
// Precondition: pegNumber is 1, 2, or 3.
// Postcondition: The return value is the number
// of rings on the specified peg.

int getTopDiameter(int pegNumber)
// Precondition: pegNumber is 1, 2, or 3.
// Postcondition: If countRings(pegNumber) > 0,
// then the return value is the diameter of the top
// ring on the specified peg; otherwise the return
// value is zero.

void move(int startPeg, int endPeg)
// Precondition: startPeg is a peg number
// (1, 2, or 3), and coutRings(startPeg) > 0;
// endPeg is a different peg number (not equal
// to startPeg), and if endPeg has at least one
// ring, then getTopDiameter(startPeg) is
// less than getTopDiameter(endPeg).
// Postcondition: The top ring has been moved
// from startPeg to endPeg.
```

Also include a method so that a `Towers` object may be displayed easily.

Use the `Towers` object in a program that allows a child to play Towers of Hanoi. Make sure that you don't allow the child to make any illegal moves.



At game start | After 1 move | After 2 moves | After 3 moves



After 4 moves | After 5 moves | After 6 moves | After 7 moves