Chapter 1

Bags

Contents

Prerequisites

Appendix C Creating Classes from Other Classes
Appendix D Designing Classes

Objectives

After studying this chapter, you should be able to

- Describe the concept of an abstract data type (ADT)
- Describe the ADT bag
- Use the ADT bag in a Java program

This chapter builds on the concepts of encapsulation and data abstraction presented in Appendix D, and it develops the notion of an abstract data type. As you probably know, a **data type** such as `int` or `double` is a group of values and operations on those values that is defined within a specific programming language. In contrast, an **abstract data type**, or **ADT**, is a specification for a group of values and the operations on those values that is defined conceptually and independently of any programming language. A **data structure** is an implementation of an ADT within a programming language.

This chapter also begins to generalize the idea of grouping objects. A **collection** is an object that groups other objects and provides various services to its client. In particular, a typical collection enables a client to add, remove, retrieve, and query the objects it represents. Various collections exist for different purposes. Their behaviors

are specified abstractly and can differ in purpose according to the collection. Thus, a collection is an abstraction and is an abstract data type. However, an ADT is not necessarily a collection.

To provide an example of a collection and of an abstract data type, we will specify and use the ADT bag. In doing so we will provide a Java interface for our bag. Knowing just this interface, you will be able to use a bag in a Java program. You do not need to know how the entries in the bag are represented or how the bag operations are implemented. Indeed, your program will not depend on these specifics. As you will see, this important program characteristic is what data abstraction is all about.

The Bag

1.1 Imagine a paper bag, a reusable cloth bag, or even a plastic bag. People use bags when they shop, pack a lunch, or eat potato chips. Bags contain things. In everyday language, a bag is a kind of container. In Java, however, a **container** is an object whose class extends the standard class `Container`. Such containers are used in graphics programs. Rather than being considered a container, a **bag** in Java is a kind of collection.

What distinguishes a bag from other collections? A bag doesn't do much more than contain its items. It doesn't order them in a particular way, nor does it prevent duplicate items. Most of its behaviors could be performed by other kinds of collections. While describing the behaviors for the collection that we'll design in this chapter, let's keep in mind that we are specifying an abstraction inspired by an actual physical bag. For example, a paper bag holds things of various dimensions and shapes in no particular order and without regard for duplicates. Our abstract bag will hold unordered and possibly duplicate objects, but let's insist that these objects have the same or related types.

Note: A bag is a finite collection of objects in no particular order. A bag can contain duplicate items.

A Bag's Behaviors

1.2 Since a bag contains a finite number of objects, reporting how many objects it contains could be one of a bag's behaviors:

Get the number of items currently in the bag

Two related behaviors detect whether a bag is full or empty:

See whether the bag is full
See whether the bag is empty

1.3 We should be able to add and remove objects:

Add a given object to the bag
Remove an unspecified object from the bag
Remove an occurrence of a particular object from the bag, if possible
Remove all objects from the bag

While you hope that the bagger at the grocery store does not toss six cans of soup into a bag on top of your bread and eggs, our add operation does not indicate where in the bag an object should go. Remember that a bag does not order its contents. Likewise, the first remove operation just removes any object it can. This operation is like reaching into a grab bag and pulling something out. On the other hand, the second remove operation looks for a particular item in the bag. If you find it, you take it out. If the bag contains several equal objects that satisfy your search, you remove any one of

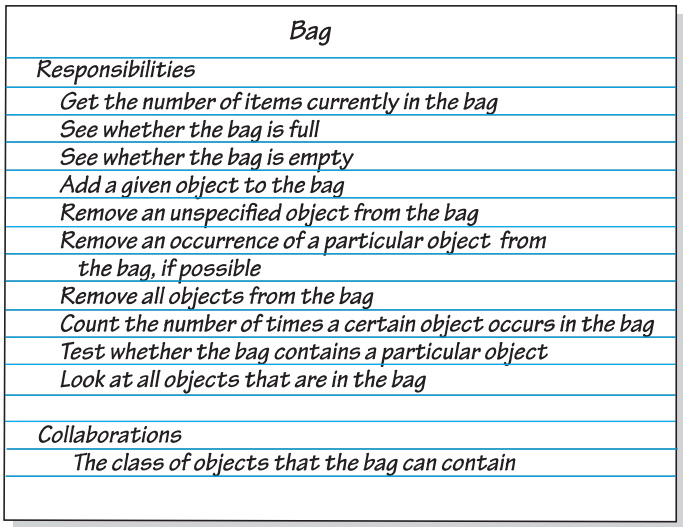
them. If you can't find the object in the bag, you can't remove it, and you just say so. Finally, the last remove operation simply empties the bag of all objects.

1.4 How many cans of dog food did you buy? Did you remember to get anchovy paste? Just what is in that bag? The answers to these questions can be answered by the following operations:

Count the number of times a certain object occurs in the bag
Test whether the bag contains a particular object
Look at all objects that are in the bag

We have enough behaviors for now. At this point, we would have written all 10 behaviors on a piece of paper or on the class-responsibility-collaboration (CRC) card pictured in Figure 1-1, as suggested in Appendix D.

FIGURE 1-1 A CRC card for a class `Bag`



1.5 Since a bag is an abstract data type, we only describe its data and specify its operations. We do not indicate how to store the data or how to implement its operations. Don't think about arrays, for example. You first need to clearly know what the bag operations do: Focus on *what* the operations do, not on *how* they do them. That is, you need a detailed set of specifications before you can use a bag in a program. In fact, you should specify the bag operations before you even decide on a programming language.

Note: Since an abstract data type, or ADT, describes a data organization independently of a programming language, you have a choice of programming languages for its implementation.

Specifying a Bag


VideoNote
Designing an ADT

Before we can implement a bag in Java, we need to describe its data and specify in detail the methods that correspond to the bag's behaviors. We'll name the methods, choose their parameters, decide their return types, and write comments to fully describe their effect on the bag's data. Our eventual goal, of course, is to write a Java header and comments for each method, but first we will express the methods in pseudocode and then in Unified Modeling Language (UML) notation.

1.6 The first behavior on our CRC card gives rise to a method that returns a count of the current number of entries in the bag. The corresponding method has no parameters and returns an integer. In pseudocode, we have the following specification:

```
// Returns the current number of entries in the bag.
getCurrentSize()
```

We can express this method using UML as

```
+getCurrentSize(): integer
```

and add this line to a class diagram.

We can test whether the bag is full or empty by using two boolean-valued methods, again without parameters. Their specifications in pseudocode and UML are

```
// Returns true if the bag is full.
isFull()

// Returns true if the bag is empty.
isEmpty()
```

and

```
+isFull(): boolean
+isEmpty(): boolean
```

We add these two lines to our class diagram.

1.7 We now want to add a given object to the bag. We can name the method `add` and give it a parameter to represent the new entry. We could write the following pseudocode:

```
// Adds a new entry to the bag.
add(newEntry)
```

We might be tempted to make `add` a void method, but if the bag is full, we cannot add a new entry to it. What should we do in this case?

Design Decision: What should the method `add` do when it cannot add a new entry?

Here are two options that we can take when `add` cannot complete its task:

- Do nothing. We cannot add another item, so we ignore it and leave the bag unchanged.
- Leave the bag unchanged, but signal the client that the addition is impossible.

The first option is easy, but it leaves the client wondering what happened. Of course, we could state as a precondition of `add` that the bag must not already be full. Then the client has the responsibility to avoid adding a new entry to a full bag.

The second option is the better one, and it is not too hard to specify or implement. How can we indicate to the client whether the addition was successful? The standard Java interface `Collection` specifies that an exception should occur if the addition is not successful. We will leave this approach for later and use another way. Displaying an error message is not a good choice, as you should let the client dictate all written output. Since the addition is either successful or not, we can simply have the method `add` return a boolean value.

Thus, we can specify the method `add` in UML as

```
+add(newEntry: T): boolean
```

where `newEntry`'s data type is the generic type[1] `T`.

1. Appendix C reviews generic types in Java.

Question 1 Suppose `aBag` represents an empty bag that has a finite capacity. Write some pseudocode statements to add user-supplied strings to the bag until it becomes full.

1.8 Three behaviors involve removing entries from a bag: remove all entries, remove any one entry, and remove a particular entry. Suppose we name the methods and any parameters and specify them in pseudocode as follows:

```
// Removes all entries from the bag.
clear()

// Removes one unspecified entry from the bag.
remove()

// Removes one occurrence of a particular entry from the bag, if possible.
remove(anEntry)
```

What return types are these methods?

1.9 The method `clear` can be a void method: We just want to empty the bag, not retrieve any of its contents. Thus, we write

```
+clear(): void
```

in UML.

If the first `remove` method removes an entry from the bag, the method can easily return the object it has removed. Its return type is then the generic type `T`. In UML, we have

```
+remove(): T
```

Notice that we can respond to an attempt to remove an object from an empty bag by returning `null`.

The second `remove` method won't be able to remove a particular entry from the bag if the bag does not contain that entry. We could have the method return a boolean value, much as `add` does, so it can indicate success or not. Or the method could return either the removed object or `null` if it can't remove the object. Here are the specifications for these two possible versions of the method in UML—we must choose one:

```
+remove(anEntry: T): boolean
```

or

```
+remove(anEntry: T): T
```

If `anEntry` equals an entry in the bag, the first version of this method would remove that entry and return true. Even though the method would not return the removed entry, the client would have the method's argument, `anEntry`, which is equal to the removed entry. We will choose this first version, to be consistent with the interface `Collection`.

Question 2 Is it legal to have both versions of `remove(anEntry)`, which were just described, in one class? Explain.

Question 3 Is it legal to have two versions of `remove`, one that has no parameter and one that has a parameter, in the same class? Explain.

Question 4 Given the full bag `aBag` that you created in Question 1, write some pseudocode statements that remove and display all of the strings in the bag.

1.10 The remaining behaviors do not change the contents of the bag. One of these behaviors counts the number of times a given object occurs within the bag. We specify it first in pseudocode and then in UML, as follows:

```
// Counts the number of times a given entry appears in the bag.
getFrequencyOf(anEntry)
```

```
+getFrequencyOf(anEntry: T): integer
```

Another method tests whether the bag contains a given object. Its specifications in pseudocode and UML are

```
// Tests whether the bag contains a given entry.
contains(anEntry)
```

```
+contains(anEntry: T): boolean
```

Question 5 Given the full bag `aBag` that you created in Question 1, write some pseudocode statements to find the number of times, if any, that the string `"Hello"` occurs in `aBag`.

1.11 Finally, we want to look at the contents of the bag. Rather than providing a method that displays the entries in the bag, we will define one that returns an array of these entries. The client is then free to display any or all of them in any way desired. Here are the specifications for our last method:

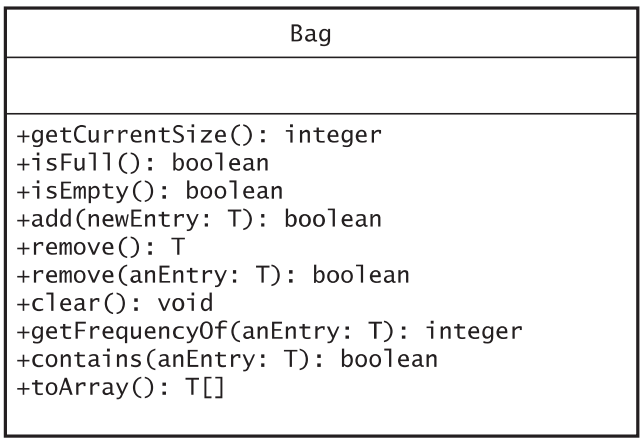
```
// Looks at all entries in the bag.
toArray()
```

```
+toArray(): T[]
```

When a method returns an array, it usually should define a new one to return. We will note that detail for this method.

1.12 As we developed the previous specifications for the bag's methods, we represented them using UML notation. Figure 1-2 shows the result of doing so.

FIGURE 1-2 UML notation for the class `Bag`



Notice that the CRC card and the UML do not reflect all of the details, such as assumptions and unusual circumstances, that we mentioned in our previous discussion. However, after you have

identified such conditions, you should specify how your methods will behave under each one. You should write down your decisions about how you want your methods to behave, as we have done in the following table. Later, you can incorporate these informal descriptions into the Java comments that document your methods.

ABSTRACT DATA TYPE: BAG

DATA

- A finite number of objects, not necessarily distinct, in no particular order, and having the same data type
- The number of objects in this collection

OPERATIONS

PSEUDOCODE	UML	DESCRIPTION
`getCurrentSize()`	`+getCurrentSize(): integer`	Task: Reports the current number of objects in the bag. Input: None. Output: The number of objects currently in the bag.
`isFull()`	`+isFull(): boolean`	Task: Sees whether the bag is full. Input: None. Output: True or false according to whether the bag is full.
`isEmpty()`	`+isEmpty(): boolean`	Task: Sees whether the bag is empty. Input: None. Output: True or false according to whether the bag is empty.
`add(newEntry)`	`+add(newEntry: T): boolean`	Task: Adds a given object to the bag. Input: `newEntry` is an object. Output: True or false according to whether the addition succeeds.
`remove()`	`+remove(): T`	Task: Removes an unspecified object from the bag, if possible. Input: None. Output: Either the removed object, if the removal was successful, or `null`.

`remove(anEntry)`	`+remove(anEntry: T): boolean`	Task: Removes an occurrence of a particular object from the bag, if possible. Input: `anEntry` is an object. Output: True or false according to whether the removal succeeds.
`clear()`	`+clear(): void`	Task: Removes all objects from the bag. Input: None. Output: None.
`getFrequencyOf(anEntry)`	`+getFrequencyOf(anEntry: T): integer`	Task: Counts the number of times an object occurs in the bag. Input: `anEntry` is an object. Output: The integer number of times `anEntry` occurs in the bag.
`contains(anEntry)`	`+contains(anEntry: T): boolean`	Task: Tests whether the bag contains a particular object. Input: `anEntry` is an object. Output: True or false according to whether `anEntry` occurs in the bag.
`toArray()`	`+toArray(): T[]`	Task: Looks at all objects in the bag. Input: None. Output: A new array of entries currently in the bag.

Design Decision: What should happen when an unusual condition occurs?

You as class designer need to make decisions about how to treat unusual conditions and include these decisions in your specifications. The documentation for the ADT bag should reflect both these decisions and the details in the previous discussion.

In general, you can address unusual situations in several ways. Your method could

- Assume that the invalid situations will not occur. This assumption is not as naive as it might sound. A method could state as an assumption—that is, a precondition—restrictions to which a client must adhere. It is then up to the client to check that the precondition is satisfied before invoking the method. For example, a precondition for the method `remove` might be that the bag is not empty. Notice that the client can use other methods of the ADT bag, such as `isEmpty` and `getCurrentSize`, to help with this task. As long as the client obeys the restriction, the invalid situation will not occur.

- Ignore the invalid situations. A method could simply do nothing when given invalid data. Doing absolutely nothing, however, leaves the client without knowledge of what happened.
- Guess at the client's intention. Like the previous option, this choice can cause problems for the client.
- Return a value that signals a problem. For example, if a client tries to remove an entry from an empty bag, the `remove` method could return `null`. The value returned must be something that cannot be in the bag.
- Return a boolean value that indicates the success or failure of an operation.
- Throw an exception.

Note: Throwing an exception is often a desirable way for a Java method to react to unusual events that occur during its execution. The method can simply report a problem without deciding what to do about it. The exception enables each client to do what is needed in its own particular situation. However, the method invocation in the client must appear within a `try` block. For simplicity right now, we will adopt the philosophy that methods should throw exceptions only in truly exceptional circumstances, when no other reasonable solution exists.

Note: A first draft of an ADT's specifications often overlooks or ignores situations that you really need to consider. You might intentionally make these omissions to simplify this first draft. Once you have written the major portions of the specifications, you can concentrate on the details that make the specifications complete.

An Interface

1.13 As your specifications become more detailed, they increasingly should reflect your choice of programming language. Ultimately, you can write Java headers for the bag's methods and organize them into a Java interface for the class that will implement the ADT. The Java interface in Listing 1-1 contains the methods for an ADT bag and detailed comments that describe their behaviors. Recall that a class interface does not include data fields, constructors, private methods, or protected methods.

For now, the items in the bag will be objects of the same class. For example, we could have a bag of strings. To accommodate entries of any one class type, the bag methods use a generic type `T` for each entry. To give meaning to the identifier `T`, we must write `<T>` after the name of the interface. Once the actual data type is chosen within a client, the compiler will use that data type wherever `T` appears.

As you examine the interface, notice the decisions that were made to address the unusual situations mentioned in the previous segment. In particular, each of the methods `add`, `remove`, and `contains` returns a value. Since our programming language is Java, notice that one of the `remove` methods returns a reference to an entry, not the entry itself.

Although writing an interface before implementing a class is certainly not required, doing so enables you to document your specifications in a concise way. You then can use the code in the interface as an outline for the actual class. Having an interface also provides a data type for a bag that is independent of a particular class definition. The next two chapters will develop different

implementations of a class of bags. Code written with respect to an interface allows us to more easily replace one implementation of a bag with another.

LISTING 1-1 A Java interface for a class of bags

```
/**
   An interface that describes the operations of a bag of objects.
   @author Frank M. Carrano
*/
public interface BagInterface<T>
{
   /** Gets the current number of entries in this bag.
       @return the integer number of entries currently in the bag */
   public int getCurrentSize();

   /** Sees whether this bag is full.
       @return true if the bag is full, or false if not */
   public boolean isFull();

   /** Sees whether this bag is empty.
       @return true if the bag is empty, or false if not */
   public boolean isEmpty();

   /** Adds a new entry to this bag.
       @param newEntry  the object to be added as a new entry
       @return true if the addition is successful, or false if not */
   public boolean add(T newEntry);

   /** Removes one unspecified entry from this bag, if possible.
       @return either the removed entry, if the removal
               was successful, or null */
   public T remove();

   /** Removes one occurrence of a given entry from this bag,
       if possible.
       @param anEntry  the entry to be removed
       @return true if the removal was successful, or false if not */
   public boolean remove(T anEntry);

   /** Removes all entries from this bag. */
   public void clear();

   /** Counts the number of times a given entry appears in this bag.
       @param anEntry  the entry to be counted
       @return the number of times anEntry appears in the bag */
   public int getFrequencyOf(T anEntry);

   /** Tests whether this bag contains a given entry.
       @param anEntry  the entry to locate
       @return true if the bag contains anEntry, or false otherwise */
```

```
   public boolean contains(T anEntry);

   /** Creates an array of all entries that are in this bag.
       @return a newly allocated array of all the entries in the bag */
   public T[] toArray();
} // end BagInterface
```

1.14 After specifying an ADT and writing a Java interface for its operations, you should write some Java statements that use the ADT. Although we cannot execute these statements yet—after all, we have not written a class that implements `BagInterface`—we can use them to confirm or revise both our decisions about the design of the methods and the accompanying documentation. In this way, you check both the suitability and your understanding of the specifications. It is better to revise the design or documentation of the ADT now, instead of after you have written its implementation. An added benefit of doing this task carefully is that you can use these same Java statements later to test your implementation.

Question 6 Given the full bag `aBag` that you created in Question 1, write some Java statements that display all of the strings in `aBag`. Do not alter the contents of `aBag`.

Programming Tip **Write a test program before you implement a class**

Writing Java statements that test a class's methods will help you to fully understand the specifications for the methods. Obviously, you must understand a method before you can implement it correctly. If you are also the class designer, your use of the class might help you see desirable changes to your design or its documentation. You will save time if you make these revisions before you have implemented the class. Since you must write a program that tests your implementation sometime, why not get additional benefits from the task by writing it now instead of later?

Note: Although we said that the entries in a bag belong to the same class, those entries can also belong to classes related by inheritance. For example, assume `Bag` is a class that implements the interface `BagInterface`. If we create a bag of class `C` objects by writing

```
BagInterface<C> aBag = new Bag<C>();
```

`aBag` can contain objects of class `C`, as well as objects of any subclass of `C`.

The following section looks at two examples that use a bag. Later, these examples can be part of a test of your implementation.

Using the ADT Bag

1.15 Imagine that we hire a programmer to implement the ADT bag in Java, given the interface and specifications that we have developed so far. If we assume that these specifications are clear enough for the programmer to complete the implementation, we can use the ADT's operations in a

program without knowing the details of the implementation. That is, we do not need to know *how* the programmer implemented the bag to be able to use it. We only need to know *what* the ADT bag does. This section assumes that we have a Java class, `Bag`, that implements the Java interface `BagInterface` given in Listing 1-1. The simple examples demonstrate how we can use `Bag`.


VideoNote
Designing a test for an ADT

Notice that once we choose the data type of the objects to be in a bag, that data type is enclosed in brackets that follow the interface name and the class name. All entries in the bag then must have either that data type or a subtype of that data type. The compiler will enforce this restriction for us. For primitive types, you can place instances of an appropriate wrapper class into a bag. For example, instead of instances of the primitive type `int`, you could use instances of the wrapper class `Integer`.

1.16

Example: Shopping online. When you shop online, your selections are saved in a shopping cart, or bag, until you are ready to check out. The program that implements the shopping website can use the class `Bag` to maintain the shopping cart. After all, the order in which you choose items to purchase is not important. Listing 1-2 shows a simple example of such a program.

LISTING 1-2 A program that maintains a bag for online shopping

```
/**
    A class that maintains a shopping cart for an online store.
    @author Frank M. Carrano
*/
public class OnlineShopper
{
   public static void main(String[] args)
   {
      Item[] items = {new Item("Bird feeder", 2050),
                      new Item("Squirrel guard", 1547),
                      new Item("Bird bath", 4499),
                      new Item("Sunflower seeds", 1295)};
      BagInterface<Item> shoppingCart = new Bag<Item>();
      int totalCost = 0;

      // statements that add selected items to the shopping cart:
      for (int index = 0; index < items.length; index++)
      {
         Item nextItem = items[index]; // simulate getting item from
                                       // shopper
         shoppingCart.add(nextItem);
         totalCost = totalCost + nextItem.getPrice();
      } // end for

      // simulate checkout
      while (!shoppingCart.isEmpty())
         System.out.println(shoppingCart.remove());

      System.out.println("Total cost: " +
                         "\t$" + totalCost / 100 + "." +
                         totalCost % 100);
```

```
   } // end main
} // end OnlineShopper
```

Output

```
Sunflower seeds $12.95
Bird bath       $44.99
Squirrel guard  $15.47
Bird feeder     $20.50
Total cost:     $93.91
```

To keep the example simple, we create an array of `Item` objects to represent the choices made by the shopper. The class `Item`, which is available to you in this book's online resources, defines data fields for an item's description and price, accessor methods for these fields, and the method `toString`.

Initially, we create an empty bag for `Item` objects by using `Bag`'s default constructor. Notice that the data type of `shoppingCart` is `BagInterface<Item>`. This declaration obliges `shoppingCart` to receive only calls to methods declared in `BagInterface`. Moreover, we could replace the class `Bag` with another class that also implements `BagInterface` without modifying the subsequent statements in the program.

Notice the loop that adds the chosen items to the bag and the loop that removes them one at a time during checkout.

Question 7 In the previous example, a `while` loop executes during the checkout process until the bag is empty. What `for` statement could replace the `while` statement? Use only the existence of `shoppingCart`, not the array `items`.

1.17

Example: A piggy bank. You might have a piggy bank, jar, or some other receptacle to hold your spare coins. The piggy bank holds the coins but gives them no other organization. And certainly the bank can contain duplicate coins. A piggy bank is like a bag, but it is simpler, as it has only three operations: You can add a coin to the bank, remove one (you shake the bank, so you have no control over what coin falls out), or see whether the bank is empty.

Assuming that we have the class `Coin` to represent coins, we can create the class `PiggyBank` given in Listing 1-3. A `PiggyBank` object stores its coins in a bag, that is, in an instance of a class that implements the interface `BagInterface`. The `add`, `remove`, and `isEmpty` methods of `PiggyBank` each call the respective bag method to achieve their results. The class `PiggyBank` is an example of an adapter class. See Appendix C for more on adapter classes.

LISTING 1-3 A class of piggy banks

```
/**
   A class that implements a piggy bank by using a bag.
   @author Frank M. Carrano
*/
public class PiggyBank
{
   private BagInterface<Coin> coins;

   public PiggyBank()
   {
```

```
      coins = new Bag<Coin>();
   } // end default constructor

   public boolean add(Coin aCoin)
   {
      return coins.add(aCoin);
   } // end add

   public Coin remove()
   {
      return coins.remove();
   } // end remove

   public boolean isEmpty()
   {
      return coins.isEmpty();
   } // end isEmpty
} // end PiggyBank
```

1.18 Listing 1-4 provides a brief demonstration of the class `PiggyBank`. The program adds some coins to the bank and then removes all of them. Since the program does not keep a record of the coins it adds to the bank, it has no control over which coins are removed. Although the output indicates that the coins leave the bank in the opposite order from how they entered it, that order depends on the bag's implementation. We'll consider these implementations in the next chapters.

Notice that, in addition to the `main` method, the program defines another method, `addCoin`. Since `main` is static and calls `addCoin`, it must be static as well. The method `addCoin` accepts as its arguments a `Coin` object and a `PiggyBank` object. The method then adds the coin to the bank.

LISTING 1-4 A demonstration of the class `PiggyBank`

```
/**
    A class that demonstrates the class PiggyBank.
    @author Frank M. Carrano
*/
public class PiggyBankExample
{
   public static void main(String[] args)
   {
      PiggyBank myBank = new PiggyBank();

      addCoin(new Coin(1, 2010), myBank);
      addCoin(new Coin(5, 2011), myBank);
      addCoin(new Coin(10, 2000), myBank);
      addCoin(new Coin(25, 2012), myBank);
```

```
      System.out.println("Removing all the coins:");
      int amountRemoved = 0;

      while (!myBank.isEmpty())
      {
         Coin removedCoin = myBank.remove();
         System.out.println("Removed a " + removedCoin.getCoinName() +
                            ".");
         amountRemoved = amountRemoved + removedCoin.getValue();
      } // end while

      System.out.println("All done. Removed " + amountRemoved +
                         " cents.");
   } // end main

   private static void addCoin(Coin aCoin, PiggyBank aBank)
   {
      if (aBank.add(aCoin))
         System.out.println("Added a " + aCoin.getCoinName() + ".");
      else
         System.out.println("Tried to add a " + aCoin.getCoinName() +
                            ", but couldn't");
   } // end addCoin
} // end PiggyBankExample
```

Output

```
Added a PENNY.
Added a NICKEL.
Added a DIME.
Added a QUARTER.
Removing all the coins:
Removed a QUARTER.
Removed a DIME.
Removed a NICKEL.
Removed a PENNY.
All done. Removed 41 cents.
```

Note: A method can change the state of an object passed to it as an argument

You pass two arguments to the method `addCoin`: a coin and a piggy bank. Both of these arguments are references to objects that exist in the `main` method. The method `addCoin` stores copies of these references in its parameters, which, as you will recall, behave as local variables. Although `addCoin` cannot change the references, because they exist in the `main` method, it can alter the state of the referenced objects. In particular, it changes the piggy bank—that is, the `PiggyBank` object—by adding coins to it. That bank, remember, is local to `main` and is outside of `addCoin`.

Note: As soon as we implement a class of bags in the next chapters, you can actually run the programs shown in the previous listings. You just need to reconcile the class name `Bag` that these examples use with the names of the classes in the next chapters.

Question 8 Consider the program in Listing 1-4. After creating the instance `myBank` of the class `PiggyBank`, suppose that we add several unknown coins to `myBank`. Write some code that will remove coins from the bank until either you remove a penny or the bank becomes empty.

Using an ADT Is Like Using a Vending Machine

1.19 Imagine that you are in front of a vending machine, as Figure 1-3 depicts; or better yet, take a break and go buy something from one!

FIGURE 1-3 A vending machine

When you look at the front of a vending machine, you see its interface. By inserting coins and pressing buttons, you are able to make a purchase. Here are some observations that we can make about the vending machine:

- You can perform only the specific tasks that the machine's interface presents to you.
- You must understand these tasks—that is, you must know what to do to buy a soda.
- You cannot access the inside of the machine, because a locked shell encapsulates it.
- You can use the machine even though you do not know what happens inside.
- If someone replaced the machine's inner mechanism with an improved version, leaving the interface unchanged, you could still use the machine in the same way.

You, as the user of a vending machine, are like the client of the ADT bag that you saw earlier in this chapter. The observations that we just made about the user of a vending machine are similar to the following observations about a bag's client:

- The client can perform only the operations specific to the ADT bag. These operations often are declared within a Java interface.
- The client must adhere to the specifications of the operations that the ADT bag provides. That is, the programmer of the client must understand how to use these operations.
- The client cannot access the data within the bag without using an ADT operation. The principle of encapsulation hides the data representation within the ADT.
- The client can use the bag, even though the programmer does not know how the data is stored.
- If someone changed the implementation of the bag's operations, the client could still use the bag in the same way, as long as the interface did not change.

1.20 In the examples of the previous section, each bag is an instance of a class that implements the ADT bag. That is, each bag is an object whose behaviors are the operations of the ADT bag. You can think of each such object as being like the vending machine we just described. Each object encapsulates the bag's data and operations, just as the vending machine encapsulates its product (soda cans) and delivery system.

Some ADT operations have inputs analogous to the coins you insert into a vending machine. Some ADT operations have outputs analogous to the change, soda cans, messages, and warning lights that a vending machine provides.

Now imagine that you are the designer of the front, or interface, of the vending machine. What can the machine do, and what should a person do to use the machine? Will it help you or hinder you to think about how the soda cans will be stored and transported within the machine? We maintain that you should ignore these aspects and focus solely on how someone will use the machine—that is, you focus on designing the interface. Ignoring extraneous details makes your task easier and increases the quality of your design.

Recall that abstraction as a design principle asks you to focus on *what* instead of *how*. When you design an ADT, and ultimately a class, you use data abstraction to focus on what you want to do with or to the data, without worrying about how you will accomplish these tasks. We practiced data abstraction at the beginning of this chapter when we designed the ADT bag. As we chose the methods that a bag would have, we did not consider how we would represent the bag. Instead, we focused on what each method should do.

Ultimately, we wrote a Java interface that specified the methods in detail. We were then able to write a client that used the bag, again without knowledge of its implementation. If someone wrote the implementation for us, our program would presumably run correctly. If someone else gave us a better implementation, we could use it without changing our already-written client. This feature of the client is a major advantage of abstraction.

Java Class Library: The Interface Set

As we mentioned at the end of Appendix B, the Java Class Library is a collection of classes and interfaces that Java programmers use as a matter of course. From time to time, we will present members of the Java Class Library that are like or relevant to our current discussion. The **Java Collections Framework** is a subset of this library that provides us with a uniform way of representing and working with collections. Many of the classes and interfaces in the Java Class Library that we will note are a part of this framework, although we usually will not point out this fact.

1.21 The ADT **set** is a bag that does not allow duplicate entries. Although we leave the specification and implementation of the set as exercises in this and subsequent chapters, we do want to present the standard interface `Set`, which belongs to the package `java.util` within the Java Class Library. Sets that adhere to the specifications of this interface do not contain a pair of objects `x` and `y` such that `x.equals(y)` is true.

The following method headers declared in the interface `Set` are similar to the methods within our `BagInterface`. The differences between a method in `Set` and a corresponding method in `BagInterface` are highlighted.

```
public boolean add(T newEntry)
public boolean remove(Object anEntry)
public void clear()
public boolean contains(Object anEntry)
public boolean isEmpty()
public int size()
public Object[] toArray()
```

Each of the interfaces `Set` and `BagInterface` declare methods that are not in the other.

Chapter Summary

- An abstract data type, or ADT, is a specification of a data set and the operations on that data. This specification does not indicate how to store the data or how to implement the operations, and it is independent of any programming language.
- When you use data abstraction to design an ADT, you focus on what you want to do with or to the data without worrying about how you will accomplish these tasks. That is, you ignore the details of how you represent data and how you manipulate it.
- The manifestation of the ADT in a programming language encapsulates the data and operations. As a result, the particular data representations and method implementations are hidden from the client.
- A collection is an object that holds a group of other objects.
- A bag is a finite collection whose entries are in no particular order.
- A client manipulates or accesses a bag's entries by using only the operations defined for the ADT bag.
- When you add an object to a bag, you cannot indicate where in the bag it will be placed.
- You can remove from a bag an object having either a given value or one that is unspecified. You also can remove all objects from a bag.
- A bag can report whether it contains a given object. It can also report the number of times a given object occurs within its contents.
- A bag can tell you the number of objects it currently contains and can provide an array of those objects.
- Carefully specify the methods for a proposed class before you begin to implement them, using tools such as CRC cards and UML notation.
- After designing a draft of an ADT, confirm your understanding of the operations and their design by writing some pseudocode that uses the ADT.
- You should specify the action a method should take if it encounters an unusual situation.
- Writing a Java interface is a way to organize a specification for an ADT.
- Writing a program that tests a class before it is defined is a way to see whether you fully understand and are satisfied with the specification of the class's methods.

Programming Tip

- Writing Java statements that test a class's methods will help you to fully understand the specifications for the methods. Obviously, you must understand a method before you can implement it correctly. If you are also the class designer, your use of the class might help you see desirable changes to your design or its documentation. You will save time if you make these revisions before you have implemented the class. Since you must write a program that tests your implementation sometime, why not get additional benefits from the task by writing it now instead of later?

Exercises

1. Specify each method of the class `PiggyBank`, as given in Listing 1-3, by stating the method's purpose; by describing its parameters; and by writing preconditions, postconditions, and a pseudocode version of its header. Then write a Java interface for these methods that includes `javadoc`-style comments.

2. Suppose that `groceryBag` is a bag filled to its capacity with 10 strings that name various groceries. Write Java statements that remove and count all occurrences of `"soup"` in `groceryBag`. Do not remove any other strings from the bag. Report the number of times that `"soup"` occurred in the bag. Accommodate the possibility that `groceryBag` does not contain any occurrence of `"soup"`.

3. Given `groceryBag`, as described in Exercise 2, what effect does the operation `groceryBag.toArray()` have on `groceryBag`?

4. Given `groceryBag`, as described in Exercise 2, write some Java statements that create an array of the distinct strings that are in this bag. That is, if `"soup"` occurs three times in `groceryBag`, it should only appear once in your array. After you have finished creating this array, the contents of `groceryBag` should be unchanged.

5. The *union* of two collections consists of their contents combined into a new collection. Add a method `union` to the interface `BagInterface` for the ADT bag that returns as a new bag the union of the bag receiving the call to the method and the bag that is the method's one argument. Include sufficient comments to fully specify the method.

   Note that the union of two bags might contain duplicate items. For example, if object *x* occurs five times in one bag and twice in another, the union of these bags contains *x* seven times. Specifically, suppose that `bag1` and `bag2` are `Bag` objects, where `Bag` implements `BagInterface`; `bag1` contains the `String` objects `a`, `b`, and `c`; and `bag2` contains the `String` objects `b`, `b`, `d`, and `e`. After the statement

   ```
   BagInterface<String> everything = bag1.union(bag2);
   ```

   executes, the bag `everything` contains the strings `a`, `b`, `b`, `b`, `c`, `d`, and `e`. Note that `union` does not affect the contents of `bag1` and `bag2`.

6. The *intersection* of two collections is a new collection of the entries that occur in both collections. That is, it contains the overlapping entries. Add a method `intersection` to the interface `BagInterface` for the ADT bag that returns as a new bag the intersection of the bag receiving the call to the method and the bag that is the method's one argument. Include sufficient comments to fully specify the method.

   Note that the intersection of two bags might contain duplicate items. For example, if object *x* occurs five times in one bag and twice in another, the intersection of these bags contains *x* twice. Specifically, suppose that `bag1` and `bag2` are `Bag` objects, where `Bag` implements `BagInterface`; `bag1` contains the `String` objects `a`, `b`, and `c`; and `bag2` contains the `String` objects `b`, `b`, `d`, and `e`. After the statement

   ```
   BagInterface<String> commonItems = bag1.intersection(bag2);
   ```

   executes, the bag `commonItems` contains only the string `b`. If `b` had occurred in `bag1` twice, `commonItems` would have contained two occurrences of `b`, since `bag2` also contains two occurrences of `b`. Note that `intersection` does not affect the contents of `bag1` and `bag2`.

7. The *difference* of two collections is a new collection of the entries that would be left in one collection after removing those that also occur in the second. Add a method `difference` to the interface `BagInterface` for the ADT bag that returns as a new bag the difference of the bag receiving the call to the method and the bag that is the method's one argument. Include sufficient comments to fully specify the method.

Note that the difference of two bags might contain duplicate items. For example, if object *x* occurs five times in one bag and twice in another, the difference of these bags contains *x* three times. Specifically, suppose that `bag1` and `bag2` are `Bag` objects, where `Bag` implements `BagInterface`; `bag1` contains the `String` objects `a`, `b`, and `c`; and `bag2` contains the `String` objects `b`, `b`, `d`, and `e`. After the statement

```
BagInterface leftOver1 = bag1.difference(bag2);
```

executes, the bag `leftOver1` contains the strings `a` and `c`. After the statement

```
BagInterface leftOver2 = bag2.difference(bag1);
```

executes, the bag `leftOver2` contains the strings `b`, `d`, and `e`. Note that `difference` does not affect the contents of `bag1` and `bag2`.

8. Write code that accomplishes the following tasks: Consider two bags that can hold strings. One bag is named `letters` and contains several one-letter strings. The other bag is empty and is named `vowels`. One at a time, remove a string from `letters`. If the string contains a vowel, place it into the bag `vowels`; otherwise, discard the string. After you have checked all of the strings in `letters`, report the number of vowels in the bag `vowels` and the number of times each vowel appears in the bag.

9. Write code that accomplishes the following tasks: Consider three bags that can hold strings. One bag is named `letters` and contains several one-letter strings. Another bag is named `vowels` and contains five strings, one for each vowel. The third bag is empty and is named `consonants`. One at a time, remove a string from `letters`. Check whether the string is in the bag `vowels`. If it is, discard the string. Otherwise, place it into the bag `consonants`. After you have checked all of the strings in `letters`, report the number of consonants in the bag `consonants` and the number of times each consonant appears in the bag.

PROJECTS

1. As we mentioned in Segment 1.21, a set is a special bag that does not allow duplicates.

- **a.** Specify each operation for a set of objects by stating its purpose; by describing its parameters; and by writing preconditions, postconditions, and a pseudocode version of its header. Then write a Java interface, `SetInterface<T>`, for the set. Include `javadoc`-style comments in your code.
- **b.** Suppose the class `Set<T>` implements `SetInterface<T>`. Given an empty set that is an object of `Set<String>` and an object of the class `Bag<String>` that contains several strings, write statements at the client level that create a set from the given bag.

2. Imagine a pile of books on your desk. Each book is so large and heavy that you can remove only the top one from the pile. You cannot remove a book from under another one. Likewise, you can add another book to the pile only by placing it on the top of the pile. You cannot add a book beneath another one.

If you represent books by their titles alone, design a class that you can use to track the books in the pile on your desk. Specify each operation by stating its purpose, by describing its parameters, and by writing a pseudocode version of its header. Then write a Java interface for the pile's methods. Include `javadoc`-style comments in your code.

3. A *ring* is a collection of items that has a reference to a current item. An operation—let's call it `advance`—moves the reference to the next item in the collection. When the reference reaches the last item, the next `advance` operation will move the reference back to the first item. A ring also has operations to get the current item, add an item, and remove an item. The details of where an item is added and which one is removed are up to you.

   Design an ADT to represent a ring of objects. Specify each operation by stating its purpose, by describing its parameters, and by writing a pseudocode version of its header. Then write a Java interface for a ring's methods. Include `javadoc`-style comments in your code.

4. A *shoe* of playing cards contains some number of standard decks of cards. Cards in the shoe can be shuffled together and dealt one at a time. The number of cards in the shoe can also be calculated.

   After a hand is complete, you should be able to return all cards to the shoe and shuffle them. Some card games require that the discard pile be returned to the shoe when the shoe becomes empty. Then the cards in the shoe can be shuffled. In this case, not all cards are in the shoe; some are held by the players.

   Design an ADT for a shoe, assuming that you have the class `PlayingCard`, which was described in Project 6 of the online projects for Appendix C. You do not need an ADT deck, since a deck is a shoe whose number of decks is 1.

   Specify each ADT operation by stating its purpose, by describing its parameters, and by writing a pseudocode version of its header. Then write a Java interface for a shoe's methods. Include `javadoc`-style comments in your code.

5. A bid for installing an air conditioner consists of the name of the company, a description of the unit, the performance of the unit, the cost of the unit, and the cost of installation.

   Design an ADT that represents any bid. Then design another ADT to represent a collection of bids. The second ADT should include methods to search for bids based on price and performance. Also note that a single company could make multiple bids, each with a different unit.

   Specify each ADT operation by stating its purpose, by describing its parameters, and by writing a pseudocode version of its header. Then write a Java interface for a bid's methods. Include `javadoc`-style comments in your code.

6. A *matrix* is a rectangular array of numerical values. You can add or multiply two matrices to form a third matrix. You can multiply a matrix by a scalar, and you can transpose a matrix. Design an ADT that represents a matrix that has these operations.

   Specify each ADT operation by stating its purpose, by describing its parameters, and by writing a pseudocode version of its header. Then write a Java interface for the methods of a matrix. Include `javadoc`-style comments in your code.

Answers to Self-Test Questions

1.
```
// aBag is empty
do
{
   entry = next string read from user
   aBag.add(entry)
} while (!aBag.isFull())
// aBag is full
```

2. No. The two methods have identical signatures. Recall that a method's return type is not a part of its signature. These methods have the same name and parameter list.

3. Yes. The two methods have different signatures. They are overloaded methods.

4.
```
// aBag is full
while (!aBag.isEmpty())
{
   entry = aBag.remove()
   Display entry
}
// aBag is empty
```

5.
```
Display "The string Hello occurs in aBag " + aBag.getFrequencyOf("Hello") + " times."
```

6.
```
String[] contents = aBag.toArray();
for (int index = 0; index < contents.length; index++)
   System.out.print(contents[index] + " ");
System.out.println();
```

7.
```
int itemCount = shoppingCart.getCurrentSize();
for (int counter = 0; counter < itemCount; counter++)
   System.out.println(shoppingCart.remove());
```

8.
```
boolean lookingForPenny = true;
while (!myBank.isEmpty() && lookingForPenny)
{
   Coin removedCoin = myBank.remove();
   System.out.println("Removed a " + removedCoin.getCoinName() + ".");
   if (removedCoin.getCoinName() == CoinName.PENNY)
// if (removedCoin.getValue() == 1) // ALTERNATE
   {
      System.out.println("Found a penny. All done!");
      lookingForPenny = false; // penny is found
   }
} // end while

if (lookingForPenny)
   System.out.println("No penny was found. Sorry!");
```

Chapter 2

Bag Implementations That Use Arrays

Contents

Prerequisites

Appendix D Designing Classes
Chapter 1 Bags

Objectives

After studying this chapter, you should be able to

- Implement the ADT bag by using a fixed-size array or an array that you expand dynamically
- Discuss the advantages and disadvantages of the two implementations presented

You have seen several examples of how to use the ADT bag in a program. This chapter presents two different ways—each involving an array—to implement a bag in Java. When you use an array to organize data, the implementation is said to be **array based**. You will see a completely different approach in the next chapter.

We begin by using an ordinary Java array to represent the entries in a bag. With this implementation, your bag could become full, just as a grocery bag does. We then offer another implementation that does not suffer from this problem. When you use

all of the space in an array, Java enables you to move the data to a larger array. The effect is to have an array that apparently expands to meet your needs. Thus, we can have a bag that is never full.

Using a Fixed-Size Array to Implement the ADT Bag

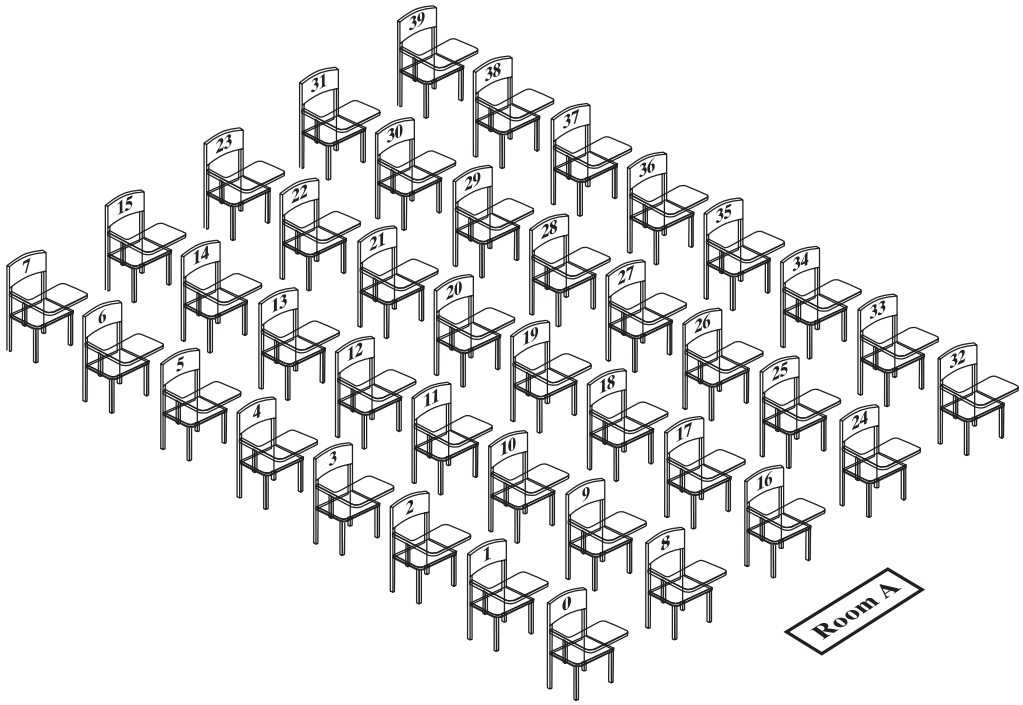
Our task is to define the methods we specified in the previous chapter when we wrote the interface `BagInterface`. We begin by using an analogy to describe how a fixed-size array could contain the entries in a bag. In doing so, we show how the `add` and `remove` methods would work. Subsequently, we present a corresponding Java implementation for the bag.

An Analogy

2.1 Imagine a classroom—call it room A—containing 40 desks in fixed positions. If a course is restricted to 30 students, 10 desks are idle and wasted. If we lift the enrollment restriction, we can accommodate only 10 more students, even if 20 more want to take the course.

An array is like this classroom, and each desk is like one array location. Suppose that we number the 40 desks in the room sequentially, beginning with zero, as Figure 2-1 illustrates. Although desks are arranged in rows in typical classrooms, we will ignore this detail and treat the desks as a one-dimensional array.

FIGURE 2-1 A classroom that contains desks in fixed positions



2.2 **Adding a new student.** Suppose that the instructor asks arriving students to occupy consecutively numbered desks. Thus, the first student who arrives at the classroom sits at desk 0; the second student sits at desk 1, and so on. The instructor's request that consecutively numbered desks be occupied is arbitrary and simply for his or her convenience. As you will see, we will fill an array of bag entries in an analogous way.

Imagine that 30 students in room A occupy the desks numbered sequentially from 0 to 29, and a new student wants to join those students. Since 40 desks are in the room, the desk numbered 30 is available. We can simply assign the new student to desk 30. When all 40 desks are occupied, we can no longer accommodate more students. The room is full.

2.3 **Removing a particular student.** Now imagine that the student in desk 5 of room A drops the course. Desk 5 stays in its fixed location within the room and will be vacant. If we still want students to sit in consecutively numbered desks, however, one student will need to move to desk 5. Since the students are not in any particular order, if the student in the highest-numbered desk moves to desk 5, no one else need move. For example, if 30 students are seated in the room in desks 0 to 29, the student in desk 29 would move to desk 5. Desks 29 and above would be vacant.

Question 1 What is an advantage of moving a student as just described so that the vacated desk does not remain vacant?

Question 2 What is an advantage of leaving the vacated desk vacant?

Question 3 If a student were to drop the course, which one could do so without forcing another to change desks?

A Group of Core Methods

2.4 The Java array-based implementation for the ADT bag incorporates some of the ideas that our classroom example illustrates. The result is the class `ArrayBag`, which implements the interface `BagInterface` that you saw in Listing 1-1 of Chapter 1. Each public method within the interface corresponds to an ADT bag operation. Recall that the interface defines a generic type `T` for the objects in a bag. We use this same generic type in the definition of `ArrayBag`.

The definition for the class `ArrayBag` could be fairly involved. The class certainly will have quite a few methods. For such classes, you should not define the entire class and then attempt to test it. Instead, you should identify a group of **core methods** to both implement and test before continuing with the rest of the class definition. By leaving the definitions of the other methods for later, you can focus your attention and simplify your task. But what methods should be part of this group? In general, such methods should be central to the purpose of the class and allow reasonable testing. We sometimes will call a group of core methods a **core group**.

When dealing with a collection such as a bag, you cannot test most methods until you have created the collection. Thus, adding objects to the collection is a fundamental operation. If the method `add` does not work correctly, testing other methods such as `remove` would be pointless. Thus, the bag's `add` method is part of the group of core methods that we implement first.

To test whether `add` works correctly, we need a method that allows us to see the bag's contents. The method `toArray` serves this purpose, and so it is a core method. The constructors are also fundamental and are in the core group. Similarly, any methods that a core method might call are part of the core group as well. For example, since we cannot add an entry to a full bag, the method `add` will need to call `isFull`.

2.5 **The core methods.** We have identified the following core methods to be a part of the first draft of the class `ArrayBag`:

- Constructors
- `public boolean add(T newEntry)`
- `public T[] toArray()`
- `public boolean isFull()`

With this core, we will be able to construct a bag, add objects to it, and look at the result. We will not implement the remaining methods until these core methods work correctly.

Note: Methods such as `add` and `remove` that can alter the underlying structure of a collection are likely to have the most involved implementations. In general, you should define such methods before the others in the class. But since we can't test `remove` before `add` is correct, we will delay implementing it until after `add` is completed and thoroughly tested.

Programming Tip: When defining a class, implement and test a group of core methods. Begin with methods that add to a collection of objects and/or have involved implementations.

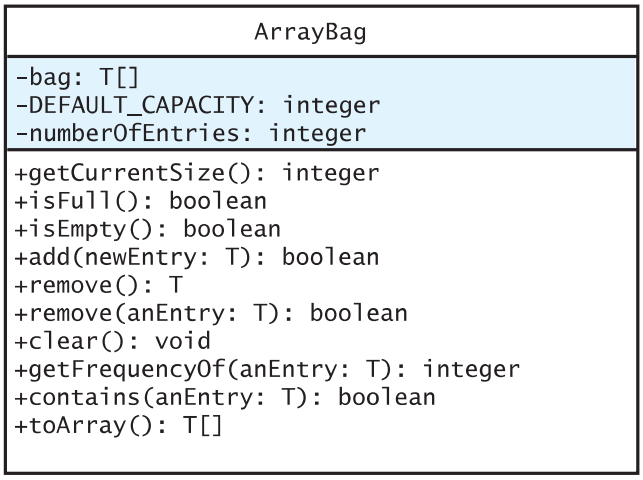
Implementing the Core Methods

2.6 **The data fields.** Before we define any of the core methods, we need to consider the class's data fields. Since the bag will hold a group of objects, one field can be an array of these objects. The length of the array defines the bag's capacity. We can let the client specify this capacity, and we can also provide a default capacity. In addition, we will want to track the current number of entries in the bag. Thus, we can define the following data fields for our class,

```
private final T[] bag;
private static final int DEFAULT_CAPACITY = 25;
private int numberOfEntries;
```

and add them to our earlier UML representation of the class in Figure 1-2 of the previous chapter. The resulting notation appears in Figure 2-2.

FIGURE 2-2 UML notation for the class `ArrayBag`, including the class's data fields



2.7 **About the constructors.** A constructor for this class must create the array `bag`. Notice that the declaration of the data field `bag` in the previous segment does not create an array. Forgetting to create an array in a constructor is a common mistake. To create the array, the constructor must specify the array's length, which is the bag's capacity. And since we are creating an empty bag, the constructor should also initialize the field `numberOfEntries` to zero.

The decision to use a generic data type in the declaration of the array bag affects how we allocate this array within the constructor. A statement such as

```
bag = new T[capacity]; // SYNTAX ERROR
```

is syntactically incorrect. You cannot use a generic type when allocating an array. Instead, we allocate an array of objects of type Object, as follows:

```
new Object[capacity];
```

However, problems arise when we try to assign this array to the data field bag. The statement

```
bag = new Object[capacity]; // SYNTAX ERROR: incompatible types
```

causes a syntax error because you cannot assign an array of type Object[] to an array of type T[]. That is, the types of the two arrays are not compatible.

A cast is necessary but creates its own problem. The statement

```
bag = (T[])new Object[capacity];
```

produces the compiler warning

```
ArrayBag.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

If you compile the class again and use the option -Xlint, the messages will be more detailed, beginning as follows:

```
ArrayBag.java:24: warning: [unchecked] unchecked cast
found   : java.lang.Object[]
required: T[]
      bag = (T[])new Object[capacity];
                 ^
```

The compiler wants you to ensure that casting each entry in the array from type Object to the generic type T is safe. Since the array has just been allocated, it contains null entries. Thus, the cast is safe, and so we instruct the compiler to ignore the warning by writing the annotation

```
@SuppressWarnings("unchecked")
```

before the offending statement. This instruction to the compiler can only precede a method definition or a variable declaration. Since the assignment

```
bag = (T[])new Object[capacity];
```

does not declare bag—bag has already been declared—we revise it as follows:

```
// the cast is safe because the new array contains null entries
@SuppressWarnings("unchecked")
T[] tempBag = (T[])new Object[capacity]; // unchecked cast
bag = tempBag;
```

Note: Suppressing compiler warnings

To suppress an unchecked-cast warning from the compiler, you precede the flagged statements with the instruction

```
@SuppressWarnings("unchecked")
```

Note that this instruction can precede only a method definition or a variable declaration.

2.8 **The constructors.** The following constructor performs the previous steps, using a capacity given as an argument:

```
/** Creates an empty bag having a given capacity.
    @param capacity  the integer capacity desired */
```

```
public ArrayBag(int capacity)
{
   numberOfEntries = 0;
   // the cast is safe because the new array contains null entries
   @SuppressWarnings("unchecked")
   T[] tempBag = (T[])new Object[capacity]; // unchecked cast
   bag = tempBag;
} // end constructor
```

The default constructor can invoke the previous one, passing it the default capacity as an argument, as follows:

```
/** Creates an empty bag whose capacity is 25. */
public ArrayBag()
{
   this(DEFAULT_CAPACITY);
} // end default constructor
```

Recall that a constructor can invoke another constructor in the same class by using the keyword `this` as a method name.

2.9 **An outline of the class.** Let's look at the class as we have defined it so far. After you complete the initial portion of the class—that is, the header, data fields, and constructors—you can add the comments and headers for the public methods simply by copying them from `BagInterface`. You then write empty bodies after each of those headers. Listing 2-1 shows the result of these steps. Our next task is to implement our three core methods.

LISTING 2-1 An outline of the class `ArrayBag`

```
/**
    A class of bags whose entries are stored in a fixed-size array.
    @author Frank M. Carrano
*/
public class ArrayBag<T> implements BagInterface<T>
{
   private final T[] bag;
   private static final int DEFAULT_CAPACITY = 25;
   private int numberOfEntries;

   /** Creates an empty bag whose initial capacity is 25. */
   public ArrayBag()
   {
      this(DEFAULT_CAPACITY);
   } // end default constructor

   /** Creates an empty bag having a given initial capacity.
       @param capacity  the integer capacity desired */
   public ArrayBag(int capacity)
   {
      numberOfEntries = 0;
      // the cast is safe because the new array contains null entries
      @SuppressWarnings("unchecked")
      T[] tempBag = (T[])new Object[capacity]; // unchecked cast
      bag = tempBag;
   } // end constructor
```

```
   /** Adds a new entry to this bag.
       @param newEntry  the object to be added as a new entry
       @return true if the addition is successful, or false if not */
   public boolean add(T newEntry)
   {
      < Body to be defined >
   } // end add

   /** Retrieves all entries that are in this bag.
       @return a newly allocated array of all the entries in the bag */
   public T[] toArray()
   {
      < Body to be defined >
   } // end toArray

   /** Sees whether this bag is full.
       @return true if the bag is full, or false if not */
   public boolean isFull()
   {
      < Body to be defined >
   } // end isFull

   < Similar partial definitions are here for the remaining methods
     declared in BagInterface. >
   . . .
} // end ArrayBag
```

Design Decision: When the array `bag` is partially full, which array elements should contain the bag's entries?

When you add a first entry to an array, you typically place it in the array's first element, that is, the element whose index is 0. Doing so, however, is not a requirement, especially for arrays that implement collections. For example, some collection implementations can benefit by ignoring the array element whose index is 0 and using index 1 as the first element in the array. Sometimes you might want to use the elements at the end of the array before the ones at its beginning. For the bag, we have no reason to be atypical, and so the objects in our bag will begin at index 0 of the array.

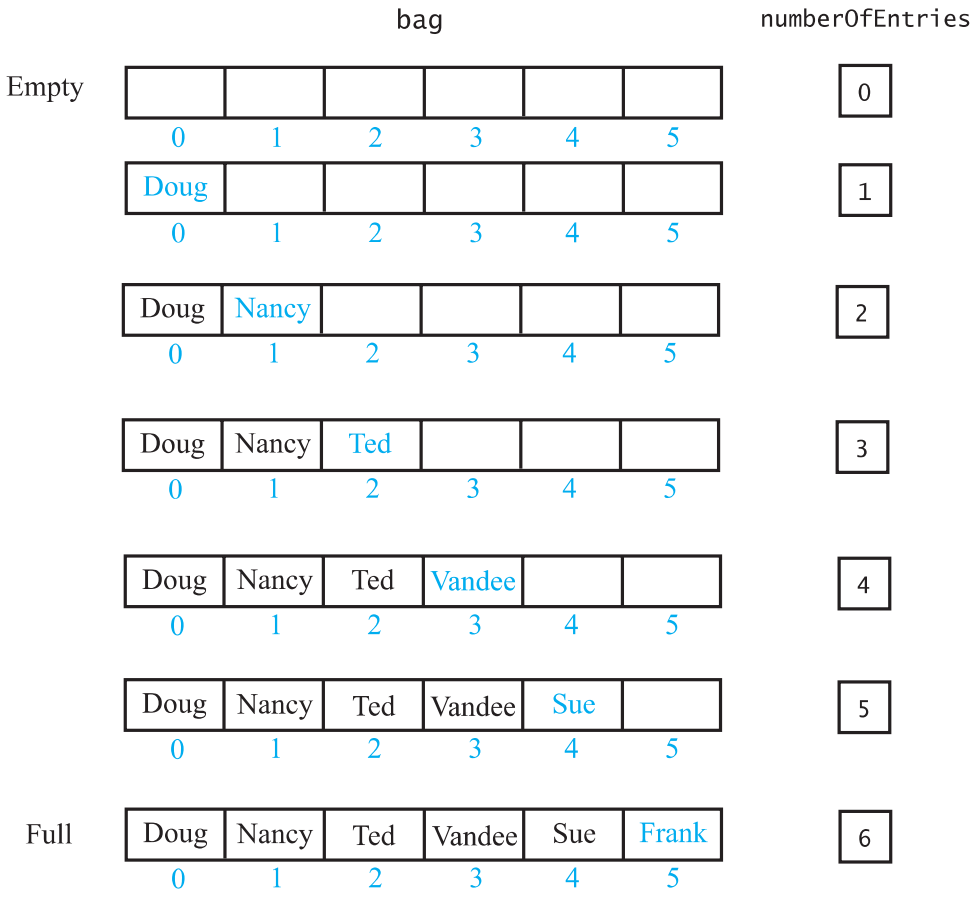
Another consideration is whether the bag's objects should occupy consecutive elements of the array. Requiring the `add` method to place objects into the array `bag` consecutively is certainly reasonable, but why should we care, and is this really a concern? We need to establish certain truths, or assertions, about our planned implementation so that the action of each method is not detrimental to other methods. For example, the method `toArray` must "know" where `add` has placed the bag's entries. Our decision now also will affect what must happen later when we remove an entry from the bag. Will the method `remove` ensure that the array entries remain in consecutive elements? It must, because for now at least, we will insist that bag entries occupy consecutive array elements.

2.10 **The method `add`.** If the bag is full, we cannot add anything to it. In that case, the method `add` should return false. Otherwise, we simply add `newEntry` immediately after the last entry in the array `bag` by writing the following statement:

```
bag[numberOfEntries] = newEntry;
```

If we are adding to an empty bag, `numberOfEntries` would be zero, and the assignment would be to `bag[0]`. If the bag contained one entry, an additional entry would be assigned to `bag[1]`, and so on. After each addition to the bag, we increase the counter `numberOfEntries`. These steps are illustrated in Figure 2-3 and accomplished by the definition of the method `add` that follows the figure.

FIGURE 2-3 Adding entries to an array that represents a bag, whose capacity is six, until it becomes full



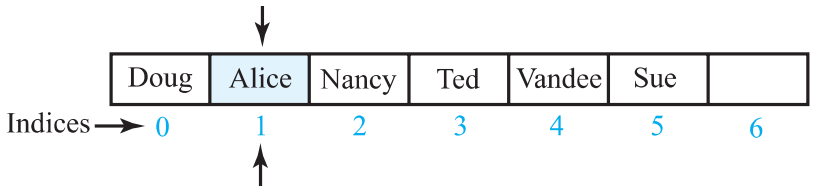

```
/** Adds a new entry to this bag.
    @param newEntry  the object to be added as a new entry
    @return true if the addition is successful, or false if not */
public boolean add(T newEntry)
{
   boolean result = true;
   if (isFull())
   {
      result = false;
   }
   else
   {  // assertion: result is true here
      bag[numberOfEntries] = newEntry;
      numberOfEntries++;
   } // end if

   return result;
} // end add
```

Notice that we call `isFull` as if it has been defined already. Had we not considered `isFull` as a core method earlier, its use now would indicate to us that it should be in the core group.

Note: The entries in a bag have no particular order. Thus, the method `add` can place a new entry into a convenient element of the array `bag`. In the previous definition of `add`, that element is the one immediately after the last element used.

Note: For simplicity, our figures and discussion portray arrays as if they actually contained objects. In reality, Java arrays contain references to objects, as Figure 2-4 illustrates.

FIGURE 2-4 An array of objects contains references to those objects



2.11 **The method `isFull`.** A bag is full when it contains as many objects as the array `bag` can accommodate. That situation occurs when `numberOfEntries` is equal to the capacity of the array. Thus, `isFull` has the following straightforward definition:

```
/** Sees whether this bag is full.
    @return true if the bag is full, or false if not */
public boolean isFull()
{
   return numberOfEntries == bag.length;
} // end isFull
```

2.12 **The method `toArray`.** The last method, `toArray`, in our initial core group retrieves the entries that are in a bag and returns them to the client within a newly allocated array. The length of this new array can equal the number of entries in the bag—that is, `numberOfEntries`—rather than the length of the array `bag`. However, we have the same problems in allocating an array that we had in defining the constructor, so we take the same steps as for the constructor.

After `toArray` creates the new array, a simple loop can copy the references in the array `bag` to this new array before returning it. Thus, the definition of `toArray` can appear as follows:

```
/** Retrieves all entries that are in this bag.
    @return a newly allocated array of all the entries in the bag */
public T[] toArray()
{
   // the cast is safe because the new array contains null entries
   @SuppressWarnings("unchecked")
   T[] result = (T[])new Object[numberOfEntries]; // unchecked cast
   for (int index = 0; index < numberOfEntries; index++)
   {
      result[index] = bag[index];
   } // end for

   return result;
} // end toArray
```

Design Decision: Should the method `toArray` return the array `bag` instead of a copy?

Suppose that we define `toArray` as follows:

```
public String[] toArray()
{
   return bag;
} // end toArray
```

This simple definition would certainly return an array of the bag's contents to a client. For example, the statement

```
String[] bagArray = myBag.toArray();
```

provides a reference to an array of the entries in `myBag`. A client could use the variable `bagArray` to display the contents of `myBag`.

The reference `bagArray`, however, is to the array `bag` itself. That is, `bagArray` is an alias for the private instance variable `bag` within the object `myBag`, and therefore it gives the client direct access to this private data. Thus, a client could change the contents of the bag without calling the class's public methods. For instance, if `myBag` is the full bag pictured in Figure 2-3, the statement

```
bagArray[2] = null;
```

would change the entry Ted to `null`. Although this approach might sound good to you if the intent is to remove Ted from the bag, doing so would destroy the integrity of the bag. In particular, the entries in the array `bag` would no longer be consecutive, and the count of the number of entries in the bag would be incorrect.

Programming Tip

A class should not return a reference to an array that is a private data field.

Note: A variable whose declared data type is `Object` can reference an object of any data type. A collection whose entries are referenced by variables of type `Object` can contain objects of various unrelated classes. In contrast, a variable having a generic data type can reference only an object of specific data types. A collection whose entries are referenced by variables of a generic type can contain only objects of classes related by inheritance. Generics enable you to restrict the data types of the entries in your collections.

Question 4 In the previous method `toArray`, does the value of `numberOfEntries` equal `bag.length` in general?

Question 5 Suppose that the previous method `toArray` gave the new array `result` the same length as the array `bag`. How would a client get the number of entries in the returned array?

Question 6 Suppose that the previous method `toArray` returned the array `bag` instead of returning a new array such as `result`. If `myBag` is a bag of five entries, what effect would the following statements have on the array `bag` and the field `numberOfEntries`?

```
Object[] bagArray = myBag.toArray();
bagArray[0] = null;
```

Question 7 The body of the method `toArray` could consist of one `return` statement if you call the method `Arrays.copyOf`. Make this change to `toArray`.

Testing the Core Methods

2.13 **Getting ready.** Now that we have defined the three core methods, we can test them. But what about the other methods in `BagInterface`? Since `ArrayBag`—as given in Listing 2-1—implements `BagInterface`, Java's syntax checker will look for a definition of each method declared in this interface. Should we wait until we complete their definitions to begin testing? Absolutely not! Testing methods as you write them makes finding logical errors easier. However, instead of writing a complete implementation of each method in `BagInterface`, we can provide incomplete definitions of the methods we choose to temporarily ignore.

An incomplete definition of a method is called a **stub**. The stub needs only to keep the syntax checker happy. For example, for each method that returns a value, you can avoid syntax errors by adding a `return` statement that returns a dummy value. Methods that return a boolean value could return true. Methods that return an object could return `null`. On the other hand, void methods can simply have an empty body.

For instance, the method `remove` ultimately will return the removed entry, so its stub must contain a `return` statement and could appear as follows:

```
public T remove()
{
   return null; // STUB
} // end remove
```

A stub for the void method `clear` could be

```
public void clear()
{
   // STUB
} // end clear
```

Note that if you plan to call a stub within your test program, the stub should report that it was invoked by displaying a message.

Programming Tip: Do not wait until you complete the implementation of an ADT before testing it. By writing stubs, which are incomplete definitions of required methods, you can begin testing early in the process.

2.14 **A test program.** Listing 2-2 shows a program to test the core methods `add`, `isFull`, and `toArray` of the class `ArrayBag`[1] at this stage of its development. Initially, the `main` method creates an empty bag by using the default constructor. Since the capacity of this bag is 25, it should not be full if you add fewer than 25 entries to it. Thus, `isFull` should return false after these additions. The program's descriptive output, in fact, indicates that the tested methods are correct.

Next in the `main` method, we consider a full bag by creating a bag whose capacity is seven and then adding seven strings to it. This time, `isFull` should return true. Again, the program's output shows that our methods are correct.

LISTING 2-2 A program that tests three core methods of the class `ArrayBag`

```
/**
   A test of the methods add, toArray, and isFull, as defined
   in the first draft of the class ArrayBag.
```

1. Note that this version of the class `ArrayBag` is available online at the book's website and is named `ArrayBag1`.

```
    @author Frank M. Carrano
*/
public class ArrayBagDemo1
{
   public static void main(String[] args)
   {
      // a bag that is not full
      BagInterface<String> aBag = new ArrayBag<String>();

      // tests on an empty bag
      testIsFull(aBag, false);

      // adding strings
      String[] contentsOfBag1 = {"A", "A", "B", "A", "C", "A"};
      testAdd(aBag, contentsOfBag1);
      testIsFull(aBag, false);

      // a bag that will be full
      aBag = new ArrayBag<String>(7);
      System.out.println("\nA new empty bag:");

      // tests on an empty bag
      testIsFull(aBag, false);

      // adding strings
      String[] contentsOfBag2 = {"A", "B", "A", "C", "B", "C", "D"};
      testAdd(aBag, contentsOfBag2);
      testIsFull(aBag, true);
   } // end main

   // Tests the method add.
   private static void testAdd(BagInterface<String> aBag,
                               String[] content)
   {
      System.out.print("Adding to the bag: ");
      for (int index = 0; index < content.length; index++)
      {
         aBag.add(content[index]);
         System.out.print(content[index] + " ");
      } // end for
      System.out.println();

      displayBag(aBag);
   } // end testAdd

   // Tests the method isFull.
   // correctResult indicates what isFull should return.
   private static void testIsFull(BagInterface<String> aBag,
                                  boolean correctResult)
   {
```

```
      System.out.print("\nTesting the method isFull with ");
      if (correctResult)
         System.out.println("a full bag:");
      else
         System.out.println("a bag that is not full:");

      System.out.print("isFull finds the bag ");
      if (correctResult && aBag.isFull())
         System.out.println("full: OK.");
      else if (correctResult)
         System.out.println("not full, but it is full: ERROR.");
      else if (!correctResult && aBag.isFull())
         System.out.println("full, but it is not full: ERROR.");
      else
         System.out.println("not full: OK.");
   } // end testIsFull

   // Tests the method toArray while displaying the bag.
   private static void displayBag(BagInterface<String> aBag)
   {
      System.out.println("The bag contains the following string(s):");
      Object[] bagArray = aBag.toArray();
      for (int index = 0; index < bagArray.length; index++)
      {
         System.out.print(bagArray[index] + " ");
      } // end for

      System.out.println();
   } // end displayBag
} // end ArrayBagDemo1
```

Output

```
Testing the method isFull with a bag that is not full:
isFull finds the bag not full: OK.
Adding to the bag: A A B A C A
The bag contains the following string(s):
A A B A C A

Testing the method isFull with a bag that is not full:
isFull finds the bag not full: OK.

A new empty bag:

Testing the method isFull with a bag that is not full:
isFull finds the bag not full: OK.
Adding to the bag: A B A C B C D
```

```
The bag contains the following string(s):
A B A C B C D

Testing the method isFull with a full bag:
isFull finds the bag full: OK.
```

2.15 Notice that, in addition to the `main` method, `ArrayBagDemo1` has three other methods. Since `main` is static and calls these other methods, they must be static as well. The method `testAdd` accepts as its arguments a bag and an array of strings. The method uses a loop to add each string in the array to the bag. The method `testIsFull` takes a bag as its argument and a boolean value that indicates the value we expect `isFull` to return if its logic is correct. Finally, the method `displayBag` takes a bag as its argument and uses the bag's method `toArray` to access its contents. Once we have an array of the bag's entries, a simple loop can display them.

Question 8 What is the result of executing the following statements within the `main` method of `BagDemo1`?

```
ArrayBag<String> aBag = new ArrayBag<String>();
displayBag(aBag);
```

Implementing More Methods

Now that we can add objects to a bag, we can implement the remaining methods, beginning with the easiest ones. We will postpone the definitions of `remove` momentarily until we see how to search a bag.

2.16 **The methods `isEmpty` and `getCurrentSize`.** The methods `isEmpty` and `getCurrentSize` have straightforward definitions, as you can see:

```
/** Sees whether this bag is empty.
    @return true if the bag is empty, or false if not */
public boolean isEmpty()
{
   return numberOfEntries == 0;
} // end isEmpty

/** Gets the current number of entries in this bag.
    @return the integer number of entries currently in the bag */
public int getCurrentSize()
{
   return numberOfEntries;
} // end getCurrentSize
```

Note: The definitions of some methods are almost as simple as the stubs you might use to define them in an early version of a class. Such is the case for the bag methods `isEmpty` and `getCurrentSize`. Although these two methods are not in our first group of core methods, they could have been. That is, we could have defined them earlier instead of writing stubs.

2.17 **The method `getFrequencyOf`.** To count the number of times a given object occurs in a bag, we count the number of times the object occurs in the array `bag`. Using a `for` loop to cycle through the array's indices from 0 to `numberOfEntries – 1`, we compare the given object to every object in the array.

Each time we find a match, we increment a counter. When the loop ends, we simply return the value of the counter. Note that we must use the method `equals` to compare objects. That is, we must write

```
anEntry.equals(bag[index])
```

and not

```
anEntry == bag[index] // WRONG!
```

We assume that the class to which the objects belong defines its own version of `equals`.

The method definition follows:

```
/** Counts the number of times a given entry appears in this bag.
    @param anEntry  the entry to be counted
    @return the number of times anEntry appears in the bag */
public int getFrequencyOf(T anEntry)
{
   int counter = 0;

   for (int index = 0; index < numberOfEntries; index++)
   {
      if (anEntry.equals(bag[index]))
      {
         counter++;
      } // end if
   } // end for

   return counter;
} // end getFrequencyOf
```

2.18 **The method `contains`.** To see whether a bag contains a given object, we once again search the array `bag`. The loop we need here is similar to the one in the method `getFrequencyOf`, but it should stop as soon as it finds the first occurrence of the desired entry. The following pseudocode describes this logic:

```
while (anEntry is not found and we have more array elements to check)
{
   if (anEntry equals the next array entry)
      anEntry is found in the array
}
```

This loop terminates under one of two conditions: Either `anEntry` has been found in the array or the entire array has been searched without success.

Here, then, is our definition of the method `contains`:

```
/** Tests whether this bag contains a given entry.
    @param anEntry  the entry to locate
    @return true if the bag contains anEntry, or false otherwise */
public boolean contains(T anEntry)
{
   boolean found = false;

   for (int index = 0; !found && (index < numberOfEntries); index++)
   {
      if (anEntry.equals(bag[index]))
      {
         found = true;
      } // end if
   } // end for

   return found;
} // end contains
```

Question 9 The method `contains` could call `getFrequencyOf` instead of executing a loop. That is, you could define the method as follows:

```
public boolean contains(T anEntry)
{
   return getFrequencyOf(anEntry) > 0;
} // end contains
```

What is an advantage and a disadvantage of this definition as compared to the one given in the previous segment?

Note: Two kinds of loops
To count how many times an entry occurs in an array, the method `getFrequencyOf` uses a loop that cycles through all of the array's entries. In fact, the body of the loop executes `numberOfEntries` times. In contrast, to indicate whether a given entry occurs in an array, the loop in the method `contains` ends as soon as the desired entry is discovered. The body of this loop executes between one and `numberOfEntries` times. You should be comfortable writing loops that execute either a definitive or a variable number of times.

2.19 **Testing the additional methods.** As you define additional methods for the class `ArrayBag`, you should test them. The program `ArrayBagDemo2`, which is available online from the book's website, focuses only on these additional methods. However, you should form a test program incrementally so that it tests all the methods you have defined so far. The tests in `ArrayBagDemo2` are performed on a bag that is not full and on a full bag, as we did in `ArrayBagDemo1`. The version of the class `ArrayBag` to date is named `ArrayBag2` within the source code available online.

Methods That Remove Entries

We have postponed the three methods that remove entries from a bag until now because one of them is somewhat difficult and involves a search much like the one we performed in the method `contains`. We begin with the two methods that are easier to define.

2.20 **The method `clear`.** The method `clear` removes all entries from a bag, one at a time. The following definition of `clear` calls the method `remove` until the bag is empty:

```
/** Removes all entries from this bag. */
public void clear()
{
   while (!isEmpty())
      remove();
} // end clear
```

Exactly which entry is removed by each cycle of the loop is unimportant. Thus, we call the `remove` method that removes an unspecified entry. Moreover, we do not save the entry that the method returns.

Note: We can write the definition of the method `clear` in terms of the as yet undefined method `remove`. However, we cannot test `clear` completely until `remove` is defined.

Question 10 Revise the definition of the method `clear` so that it does not call `isEmpty`. *Hint*: The `while` statement should have an empty body.

Question 11 Consider the following definition of `clear`:

```
public void clear()
{
   numberOfEntries = 0;
} // end clear
```

What is a disadvantage of this definition as compared to the one shown in Segment 2.20?

2.21 **Removing an unspecified entry.** The method `remove` that has no parameter removes an unspecified entry from a bag, as long as the bag is not empty. Recall from the method's specification given in the interface in Listing 1-1 of the previous chapter that the method returns the entry it removes:

```
/** Removes one unspecified entry from this bag, if possible.
    @return either the removed entry, if the removal was successful,
            or null otherwise */
public T remove()
```

If the bag is empty before the method executes, `null` is returned.

Removing an entry from a bag involves removing it from an array. Although we can access any entry in the array `bag`, the last one is easy to remove. To do so, we

- Access the entry so it can be returned
- Set the entry's array element to `null`
- Decrement `numberOfEntries`

Decrementing `numberOfEntries` causes the last entry to be ignored, meaning that it is effectively removed, even if we did not set its location in the array to `null`.

A literal translation of the previous steps into Java leads to the following definition of the method:

```
public T remove()
{
   T result = null;
   if (numberOfEntries > 0)
   {
      result = bag[numberOfEntries - 1];
      bag[numberOfEntries - 1] = null;
      numberOfEntries--;
   } // end if

   return result;
} // end remove
```

Note that this method computes `numberOfEntries - 1` three times. The following refinement avoids this repetition:

```
public T remove()
{
   T result = null;
   if (numberOfEntries > 0)
   {
      numberOfEntries--;
      result = bag[numberOfEntries];
      bag[numberOfEntries] = null;
   } // end if
```

```
        return result;
    } // end remove
```

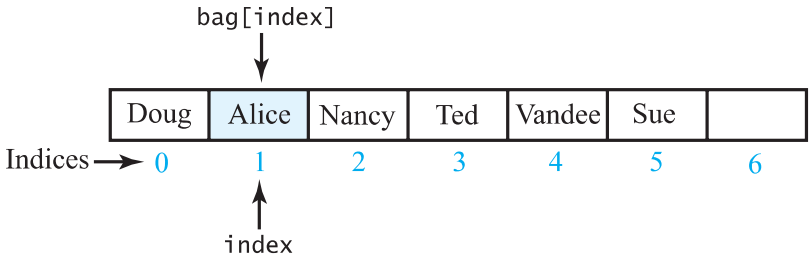
Question 12 Why does the method `remove` set `bag[numberOfEntries]` to `null`?

Question 13 The previous `remove` method removes the last entry in the array `bag`. Why might removing a different entry be more difficult to accomplish?

2.22 **Removing a given entry.** Our third method that removes an entry from the bag involves removing a given entry—call it `anEntry`. If the entry occurs more than once in the bag, we will remove only one occurrence. Exactly which occurrence is removed is unspecified. We will simply remove the first occurrence of `anEntry` that we encounter while searching for it. As we discussed in Segment 1.9 of Chapter 1, we will return either true or false, according to whether we find the entry in the bag.

Assuming that the bag is not empty, we search the array `bag` much as the method `contains` did in Segment 2.18. If `anEntry` equals `bag[index]`, we note the value of `index`. Figure 2-5 illustrates the array after a successful search.

FIGURE 2-5 The array `bag` after a successful search for the string `"Alice"`

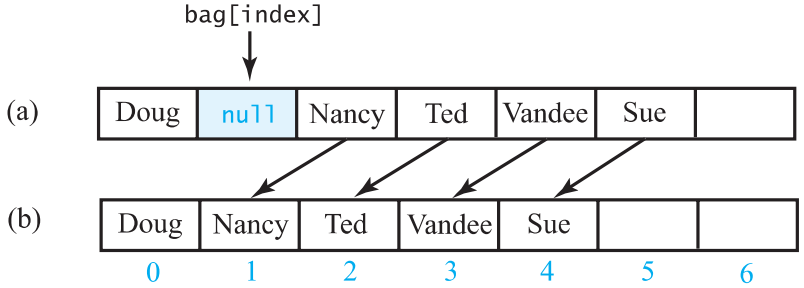


We now need to remove the entry in `bag[index]`. If we simply write

```
bag[index] = null;
```

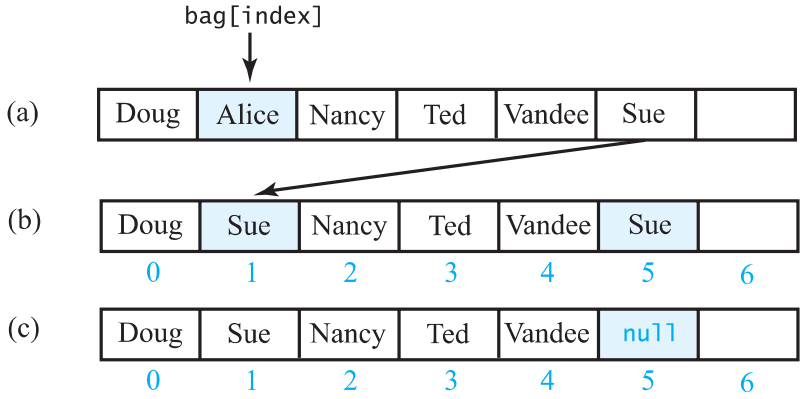
the reference in `bag[index]` to the entry will be removed, but we will have a gap in the array. That is, the contents of the bag will no longer be in consecutive array locations, as Figure 2-6a illustrates. We could get rid of that gap by shifting the subsequent entries, as shown in Figure 2-6b. This time-consuming approach is not necessary, however.

FIGURE 2-6 (a) A gap in the array `bag` after setting the entry in `bag[index]` to `null`; (b) the array after shifting subsequent entries to avoid a gap



Remember that we are not required to maintain any particular order for a bag's entries. So instead of shifting array entries after removing an entry, we can replace the entry being removed with the last entry in the array, as follows. After locating `anEntry` in `bag[index]`, as Figure 2-7a indicates, we copy the entry in `bag[numberOfEntries - 1]` to `bag[index]` (Figure 2-7b). We then replace the entry in `bag[numberOfEntries - 1]` with `null`, as Figure 2-7c illustrates, and finally we decrement `numberOfEntries`.

FIGURE 2-7 Avoiding a gap in the array while removing an entry



2.23 **Pseudocode for removing a given entry.** Let's organize our discussion by writing some pseudocode to remove the given entry, `anEntry`, from a bag that contains it:

```
Locate anEntry in the array bag; assume it occurs at bag[index]
bag[index] = bag[numberOfEntries - 1]
bag[numberOfEntries - 1] = null
Decrement the counter numberOfEntries
return true
```

This pseudocode assumes that the bag contains `anEntry`.

After we add some details to the pseudocode to accommodate the situation in which `anEntry` is not in the bag, and to avoid computing `numberOfEntries - 1` more than once, as we did in Segment 2.21, the pseudocode appears as follows:

```
Search the array bag for anEntry
if (anEntry is in the bag at bag[index])
{
   Decrement the counter numberOfEntries
   bag[index] = bag[numberOfEntries]
   bag[numberOfEntries] = null
   return true
}
else
   return false
```

2.24 **Avoiding duplicate effort.** We can easily translate this pseudocode into the Java method `remove`. However, if we were to do so, we would see much similarity between our new method and the `remove` method we wrote earlier in Segment 2.21. In fact, if `anEntry` occurs in `bag[numberOfEntries - 1]`, both `remove` methods will have exactly the same effect. To avoid this duplicate effort, both `remove` methods can call a private method that performs the removal. We can specify such a method as follows:

```
// Removes and returns the entry at a given array index.
// If no such entry exists, returns null.
private T removeEntry(int givenIndex)
```

Before we implement this private method, let's see if we can use it by revising the `remove` method in Segment 2.21. Since that method removes and returns the last entry in the array `bag`, that is, `bag[numberOfEntries - 1]`, its definition can make the call `removeEntry(numberOfEntries - 1)`. Proceeding as if `removeEntry` were defined and tested, we can define `remove` as follows:

```
/** Removes one unspecified entry from this bag, if possible.
    @return either the removed entry, if the removal was successful,
    or null otherwise */
public T remove()
{
   T result = removeEntry(numberOfEntries - 1);
   return result;
} // end remove
```

This definition looks good; let's implement the second `remove` method.

2.25 **The second `remove` method.** The first `remove` method does not search for the entry to remove, as it removes the last entry in the array. The second `remove` method, however, does need to perform a search. Rather than thinking about the details of locating an entry in an array right now, let's delegate that task to another private method, which we specify as follows:

```
// Locates a given entry within the array bag.
// Returns the index of the entry, if located, or -1 otherwise.
private int getIndexOf(T anEntry)
```

Assuming that this method is defined and tested, we can define our public method as follows:

```
/** Removes one occurrence of a given entry from this bag.
    @param anEntry  the entry to be removed
    @return true if the removal was successful, or false if not */
public boolean remove(T anEntry)
{
   int index = getIndexOf(anEntry);
   T result = removeEntry(index);
   return anEntry.equals(result);
} // end remove
```

Notice that `removeEntry` returns either the entry it removes or `null`. That is exactly what the first `remove` method needs, but the second `remove` method has to return a boolean value. Thus, in the second method we need to compare the entry we want to remove with the one `removeEntry` returns to get the desired boolean value.

Question 14 Can the `return` statement in the previous definition of `remove` be written as follows?

a. `return result.equals(anEntry);`
b. `return result != null;`

Question 15 The array `bag` in `ArrayBag` contains the entries in the bag `aBag`. If `bag` contains the strings `"A"`, `"A"`, `"B"`, `"A"`, `"C"`, why does `aBag.remove("B")` change the contents of `bag` to `"A"`, `"A"`, `"C"`, `"A"`, `null` instead of either `"A"`, `"A"`, `"A"`, `"C"`, `null` or `"A"`, `"A"`, `null`, `"A"`, `"C"`?

2.26 **The definition of the private method `removeEntry`.** Let's look back at the pseudocode we wrote in Segment 2.23 for removing a particular entry from the bag. The private method `removeEntry`

assumes that the search for the entry is done already, so we can ignore the first step of the pseudocode. The rest of the pseudocode, however, gives the basic logic for removing an entry. We can revise the pseudocode as follows:

```
// Removes and returns the entry at a given index within the arraybag.
// If no such entry exists, returns null.
if (the bag is not empty and the given index is not negative)
{
   result = bag[givenIndex]
   Decrement the counter numberOfEntries
   bag[givenIndex] = bag[numberOfEntries]
   bag[numberOfEntries] = null
   return result
}
else
   return null
```

The definition of the method `remove` given in the previous segment passes the integer returned by `getIndexOf` to `removeEntry`. Since `getIndexOf` can return –1, `removeEntry` must watch for such an argument. Thus, if the bag is not empty—that is, if `numberOfEntries` is greater than zero—and `givenIndex` is greater than or equal to zero, `removeEntry` removes the array entry at `givenIndex` by replacing it with the last entry and decrementing `numberOfEntries`. The method then returns the removed entry. If, however, the bag is empty, the method returns `null`.

The code for the method is

```
// Removes and returns the entry at a given index within the arraybag.
// If no such entry exists, returns null.
private T removeEntry(int givenIndex)
{
   T result = null;
   if (!isEmpty() && (givenIndex >= 0))
   {
      result = bag[givenIndex];                 // entry to remove
      numberOfEntries--;
      bag[givenIndex] = bag[numberOfEntries];   // replace entry with last entry
      bag[numberOfEntries] = null;              // remove last entry
   } // end if

   return result;
} // end removeEntry
```

2.27 **Locating the entry to remove.** We now need to think about locating the entry to remove from the bag so we can pass its index to `removeEntry`. The method `contains` performs the same search that we will use to locate `anEntry` within the definition of `remove`. Unfortunately, `contains` returns true or false; it does not return the index of the entry it locates in the array. Thus, we cannot simply call that method within our method definition.

Design Decision: Should the method `contains` return the index of a located entry?

Should we change the definition of `contains` so that it returns an index instead of a boolean value? No. As a public method, `contains` should not provide a client with such implementation details. The client should have no expectation that a bag's entries are in an array, since they are in no particular order. Instead of changing the specifications for `contains`, we will follow our original plan to define a private method to search for an entry and return its index.

The definition of `getIndexOf` will be like the definition of `contains`, which we recall here:

```
public boolean contains(T anEntry)
{
   boolean found = false;

   for (int index = 0; !found && (index < numberOfEntries); index++)
   {
      if (anEntry.equals(bag[index]))
      {
         found = true;
      } // end if
   } // end for

   return found;
} // end contains
```

The structure of the loop is suitable for the method `getIndexOf`, but we must save the value of `index` when the entry is found. The method will return this index instead of a boolean value.

2.28 **The definition of `getIndexOf`.** To revise the loop in `contains` for use in `getIndexOf`, we define an integer variable `where` to record the value of `index` when `anEntry` equals `bag[index]`. Thus, the definition of `getIndexOf` looks like this:

```
// Locates a given entry within the array bag.
// Returns the index of the entry, if located, or -1 otherwise.
private int getIndexOf(T anEntry)
{
   int where = -1;
   boolean found = false;

   for (int index = 0; !found && (index < numberOfEntries); index++)
   {
      if (anEntry.equals(bag[index]))
      {
         found = true;
         where = index;
      } // end if
   } // end for

   // Assertion: If where > -1, anEntry is in the array bag, and it
   // equals bag[where]; otherwise, anEntry is not in the array

   return where;

   } // end getIndexOf
```

The method `getIndexOf` returns the value of `where`. Notice that we initialize `where` to –1, which is the value to return if `anEntry` is not found.

Question 16 What `assert` statement can you add to the definition of the method `getIndexOf` just before the `return` statement to indicate the possible values that the method can return?

Question 17 Revise the definition of the method `getIndexOf` so that it does not use a boolean variable.

Aside: Thinking positively

Unlike the method `contains`, the method `getIndexOf` uses the boolean variable `found` only to control the loop and not as a return value. Thus, we can modify the logic somewhat to avoid the use of the not operator `!`.

Let's use a variable `stillLooking` instead of `found` and initialize it to true. Then we can replace the boolean expression `!found` with `stillLooking`, as you can see in the following definition of the method `getIndexOf`:

```
// Locates a given entry within the array bag.
// Returns the index of the entry, if located, or -1 otherwise.
private int getIndexOf(T anEntry)
{
   int where = -1;
   boolean stillLooking = true;

   for (int index = 0; stillLooking && (index < numberOfEntries); index++)
   {
      if (anEntry.equals(bag[index]))
      {
         stillLooking = false;
         where = index;
      } // end if
   } // end for

   return where;
} // end getIndexOf
```

If `anEntry` is found within the array, `stillLooking` is set to false to end the loop. Some programmers prefer to think positively, as in this revision, while others find `!found` to be perfectly clear.

2.29 **A revised definition for the method `contains`.** Having completed the definitions of `remove` and the private methods they call, we realize that the method `contains` can call the private method `getIndexOf`, resulting in a simpler definition than the one given in Segment 2.18. Recall that the expression `getIndexOf(anEntry)` returns an integer between 0 and `numberOfEntries - 1` if `anEntry` is in the bag, or –1 otherwise. That is, `getIndexOf(anEntry)` is greater than –1 if `anEntry` is in the bag. Thus, we can define `contains` as follows:

```
/** Tests whether this bag contains a given entry.
    @param anEntry  the entry to locate
    @return true if the bag contains anEntry, or false otherwise */
public boolean contains(T anEntry)
{
   return getIndexOf(anEntry) > -1;
} // end contains
```

Since we have changed the definition of `contains`, we should test it again. By doing so, we are also testing the private method `getIndexOf`.

Note: Both the method `contains` and the second `remove` method must perform similar searches for an entry. By isolating the search in a private method that both `contains` and `remove` can call, we make our code easier to debug and to maintain. This strategy is the same one we used when we defined the removal operation in the private method `removeEntry` that both `remove` methods call.

Programming Tip

Even though you might have written a correct definition of a method, do not hesitate to revise it if you think of a better implementation.

2.30 **Testing.** Our class `ArrayBag` is essentially complete. We can use the previously tested methods—which we assume are correct—in the tests for `remove` and `clear`. Starting with a bag that is not full, the online program `ArrayBagDemo3` removes the bag's entries until it is empty. It also includes similar tests beginning with a full bag. Finally, we should consolidate our previous tests and run them again. The source code available on the book's website identifies our test program as `ArrayBagDemo` and the complete version of the class as `ArrayBag`.

Using Array Resizing to Implement the ADT Bag

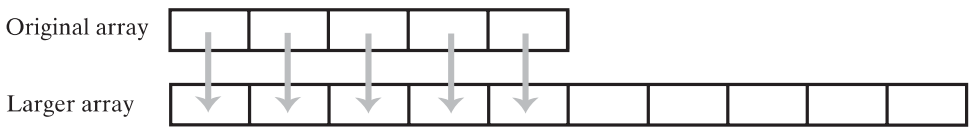
2.31 An array has a fixed size, which is chosen by either the programmer or the user before the array is created. A fixed-size array is like a classroom. If the room contains 40 desks but only 30 students, we waste 10 desks. If 40 students are taking the course, the room is full and cannot accommodate anyone else. Likewise, if we do not use all of the locations in an array, we waste memory. If we need more, we are out of luck.

Using a fixed-size array to implement the ADT bag, therefore, limits the size of the bag. When the array, and hence the bag, becomes full, the method `isFull` returns true and subsequent calls to the `add` method return false. Some applications can use a bag or other collection that has a limited capacity. For other applications, however, we need the size of a collection to grow without bound. We will now show you how a group of items can be as large as you want—within the limits of your computer's memory—but still be in an array.

Resizing an Array

2.32 **The strategy.** When a classroom is full, one way to accommodate additional students is to move to a larger room. In a similar manner, when an array becomes full, you can move its contents to a larger array. This process is called **resizing** an array. Figure 2-8 shows two arrays: an original array of five consecutive memory locations and another array—twice the size of the original array—that is in another part of the computer's memory. If you copy the data from the original smaller array to the beginning of the new larger array, the result will be like expanding the original array. The only glitch in this scheme is the name of the new array: You want it to be the same as the name of the old array. You will see how to accomplish this momentarily.

FIGURE 2-8 Resizing an array copies its contents to a larger second array

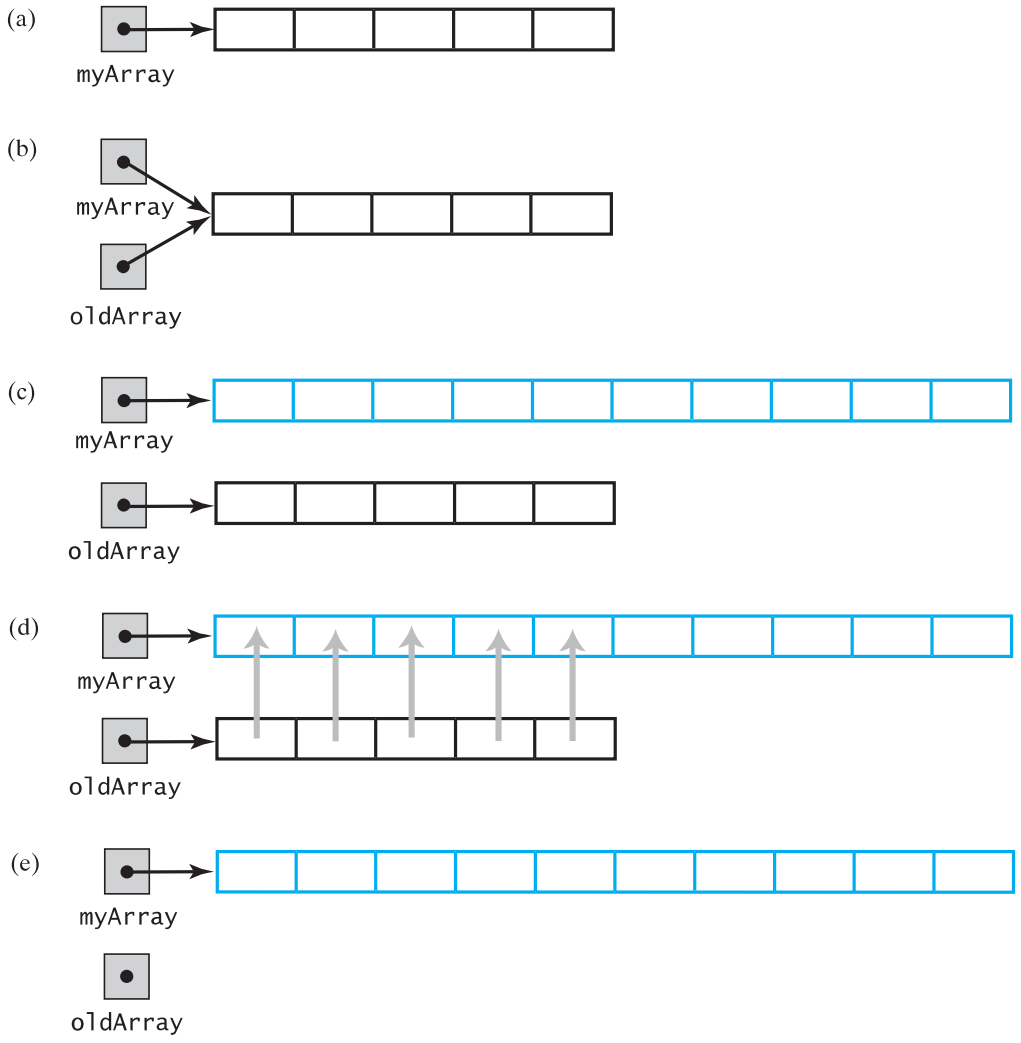


2.33 **The details.** Suppose we have an array that `myArray` references, as Figure 2-9a illustrates. We first define an alias `oldArray` that also references the array, as Figure 2-9b shows. The next step is to create a new array that is larger than the original array and let `myArray` reference this new array. As pictured in Figure 2-9c, the new array typically doubles the size of the original array. The final step

copies the contents of the original array to the new array (Figure 2-9d) and then discards the original array (Figure 2-9e). The following pseudocode summarizes these steps:

```
oldArray = myArray
myArray = a new array whose length is 2 * oldArray.length
Copy entries from the original array—oldArray–to the new array—myArray
oldArray = null // discard old array
```

FIGURE 2-9 (a) An array; (b) two references to the same array; (c) the original array variable now references a new, larger array; (d) the entries in the original array are copied to the new array; (e) the original array is discarded



Note: When an array is no longer referenced, its memory is recycled during garbage collection, just as occurs with any other object.

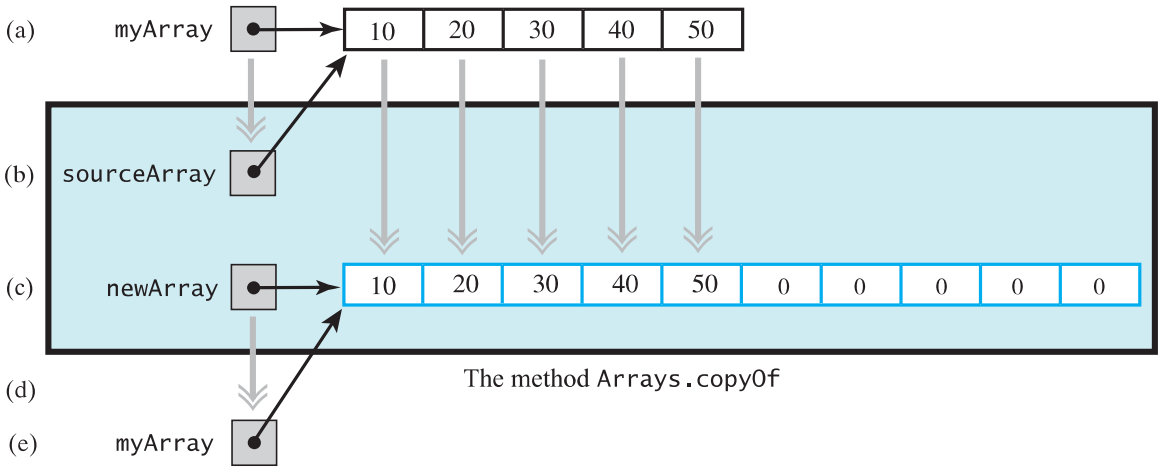
2.34 **The code.** While we could simply translate the previous pseudocode into Java, much of the work can be done by using the method `Arrays.copyOf`, which is in the Java Class Library. For example, let's work with a simple array of integers:

```
int[] myArray = {10, 20, 30, 40, 50};
```

At this point, `myArray` references the array, as Figure 2-10a shows. Next, we'll call `Arrays.copyOf`. The method's first parameter, `sourceArray`, is assigned the reference in the variable `myArray`, as Figure 2-10b implies. Next the method creates a new, larger array and copies the entries in the argument array to it (Figure 2-10c). Finally, the method returns a reference (Figure 2-10d) to the new array, and we assign this reference to `myArray` (Figure 2-10e). The following statement performs these steps:

```
myArray = Arrays.copyOf(myArray, 2 * myArray.length);
```

FIGURE 2-10 The effect of the statement `myArray = Arrays.copyOf(myArray, 2 * myArray.length);` (a) The argument array; (b) the parameter that references the argument array; (c) a new, larger array that gets the contents of the argument array; (d) the return value that references the new array; (e) the argument variable is assigned the return value



2.35 Resizing an array is not as attractive as it might first seem. Each time you expand the size of an array, you must copy its contents. If you were to expand an array by one element each time you needed additional space in the array, the process would be expensive in terms of computing time. For example, if a 50-element array is full, accommodating another entry would require you to copy the array to a 51-element array. Adding yet another entry would require that you copy the 51-element array to a 52-element array, and so on. Each addition would cause the array to be copied. If you added 100 entries to the original 50-entry array, you would copy the array 100 times.

However, expanding the array by m elements spreads the copying cost over m additions instead of just one. Doubling the size of an array each time it becomes full is a typical approach. For example, when you add an entry to a full array of 50 entries, you copy the 50-element array to a 100-element array before completing the addition. The next 49 additions then can be made quickly without copying the array. Thus, you will have copied the array only once.

Programming Tip: When increasing the size of an array, you copy its entries to a larger array. You should expand the array sufficiently to reduce the impact of the cost of copying. A common practice is to double the size of the array.

Note: Importing a class

The definition of a class that uses a class from the Java Class Library must be preceded by a `import` statement. For example, to use the class `Arrays`, you would write the following statement prior to your class definition and its descriptive comments:

```
import java.util.Arrays;
```

Some programmers replace `Arrays` in this statement with an asterisk to make all classes in the package `java.util` available to their program.

Note: To say that we "resize" an array is really a misnomer, since an array's length cannot be changed. The process of resizing an array involves creating a completely new array that contains the entries of the original array. The new array is given the name of the original array—in other words, a reference to the new array is assigned to the variable that had referenced the original array. The original array is then discarded.

Question 18 Consider the array of strings that the following statement defines:

```
String[] text = {"cat", "dog", "bird", "snake"};
```

What Java statements will increase the capacity of the array `text` by five elements without altering its current contents?

Question 19 Consider an array `text` of strings. If the number of strings placed into this array is less than its length (capacity), how could you decrease the array's length without altering its current contents? Assume that the number of strings is in the variable `size`.

A New Implementation of a Bag

2.36 **The approach.** We can revise the previous implementation of the ADT bag by resizing the array `bag` so that the bag's capacity is limited only by the amount of memory available on your computer. If we look at the outline of the class `ArrayBag` in Listing 2-1, we can see what we need to revise. Let's itemize these tasks:


VideoNote
A resizable bag

- Change the name of the class to `ResizableArrayBag` so we can distinguish between our two implementations.
- Remove the modifier `final` from the declaration of the array `bag` to enable it to be resized.
- Change the name of the constant `DEFAULT_CAPACITY` to `DEFAULT_INITIAL_CAPACITY`. Although unnecessary, this change clarifies the new purpose of the constant, since the bag's capacity will increase as necessary. Make the same change in the default constructor, which uses the constant.
- Change the names of the constructors to match the new class name.

- Revise the definition of the method `add` to always accommodate a new entry. The method will never return false.
- Revise the definition of the method `isFull` to always return false. A bag will never become full.

Revising the method `add` is the only substantial task in this list. The rest of the class will remain unchanged.

2.37 **The method `add`.** Here is the original definition of the method `add`, as it appears in Segment 2.10:

```
/** Adds a new entry to this bag.
    @param newEntry  the object to be added as a new entry
    @return true if the addition is successful, or false if not */
public boolean add(T newEntry)
{
   boolean result = true;
   if (isFull())
   {
      result = false;
   }
   else
   {  // assertion: result is true here
      bag[numberOfEntries] = newEntry;
      numberOfEntries++;
   } // end if

   return result;
} // end add
```

Since the bag will never be full, the method `isFull` will always return false. Thus, we can no longer call `isFull` to see whether the array `bag` is full. Instead, we can define a private method to both make this check and resize the array `bag`, if necessary. Let's name the method `ensureCapacity` and specify it as follows:

```
// Doubles the size of the array bag if it is full.
private void ensureCapacity()
```

Assuming that we have defined this private method, we can revise the method `add` as follows:

```
public boolean add(T newEntry)
{
   ensureCapacity();
   bag[numberOfEntries] = newEntry;
   numberOfEntries++;

   return true;
} // end add
```

2.38 **The private method `ensureCapacity`.** The array `bag` is full when `numberOfEntries` equals the array's length, `bag.length`. When that is the case, we will resize `bag` using the technique described earlier in Segment 2.34. Thus, the definition of `ensureCapacity` is straightforward:

```
// Doubles the size of the array bag if it is full.
private void ensureCapacity()
{
   if (numberOfEntries == bag.length)
      bag = Arrays.copyOf(bag, 2 * bag.length);
} // end ensureCapacity
```

2.39 **The class `ResizableArrayBag`.** Our new class is available online from the book's website. You should examine its details.

Design Decision: You might wonder about some of the decisions we made while defining the class `ResizableArrayBag`, with questions such as the following:

- Why is the method `add` a boolean method and not a void method? It always returns true!
- Why did we bother to define `isFull`? The bag is never full!
- Why did we define the private method `ensureCapacity`? Only one method, `add`, calls it!

The answers to the first two questions are the same: The class implements the interface `BagInterface`, so we followed its specifications. As a result, we have two different implementations, `ArrayBag` and `ResizableArrayBag`, each of which can be used by the same client. Our answer to the third question reflects our approach to problem solving. To implement `add`, we needed to answer two questions: When is an array full, and how do we expand a full array? Rather than risking the distraction of answering these questions while we were defining the method `add`, we chose to specify a private method to provide those answers. Admittedly, the definition of this private method turned out to be short. We could now integrate the body of the private method into that of `add`, but we have no pressing reason to do so.

Question 20 What is the definition of a constructor that you could add to the class `ResizableArrayBag` to initialize the bag to the contents of a given array?

Question 21 In the definition of the constructor described in the previous question, is it necessary to copy the entries from the argument array to the array `bag`, or would a simple assignment (`bag = contents`) be sufficient?

Question 22 What is an advantage of using an array to organize data? What is a disadvantage?

2.40 **Testing the class.** A program that tests the class `ResizableArrayBag` can create a bag whose initial capacity is small—3, for example. This choice allows us to easily test the bag's ability to increase its capacity. For instance, when the fourth item is added, the bag's capacity is doubled to 6. At the seventh addition, the capacity is doubled again, this time to 12. Such a program, `ResizableArrayBagDemo`, is available online at the book's website.

Programming Tip: A class implementing a single interface that declares the operations of an ADT should define the methods declared in the interface as its only public methods. However, the class can also define private methods and protected methods.

The Pros and Cons of Using an Array to Implement the ADT Bag

2.41 This chapter discussed two implementations of the ADT bag that use an array to store a bag's entries. An array is simple to use and enables you to access any element immediately, if you know its index. Since we know the index of the last entry in the array, removing it is easy and fast. Similarly, adding an entry at the end of the array is equally easy and fast. On the other hand, removing a particular entry, if it occurs between other entries, requires us to avoid a gap within the array. To do so, we replace the removed entry with the last entry in the array. This is an insignificant increase in execution time, as it

is overshadowed by the time it takes to locate the desired entry. We will talk more about such a search later in this book.

Using a fixed-size array limits the capacity of a bag, which is usually a disadvantage. Resizing an array dynamically enables you to increase the array's size but requires copying data. You should realize that the array entries that we copy are references, and so do not occupy much space nor take much time to move. Some languages other than Java store the data itself within the array. In such cases, moving large, complex objects can be quite time-consuming.

Note: When you use an array to implement the ADT bag,

- Adding an entry to the bag is fast
- Removing an unspecified entry is fast
- Removing a particular entry requires time to locate the entry
- Increasing the size of the array requires time to copy its entries

Chapter Summary

- You can use a Java array to define a relatively simple implementation of the ADT bag, but other implementations are possible.
- Adding an entry right after the last entry in an array does not disturb the position of existing entries. Likewise, deleting the last entry from an array does not disturb the position of existing entries.
- Because a bag does not maintain its entries in a specific order, deleting an entry does not require you to move all subsequent array entries to the next lower position. Instead, you can replace the entry that you want to delete with the last entry in the array and replace the last entry with `null`.
- Identifying and implementing a class's central, or core, methods before any others is a good strategy to use when you expect the class to be lengthy or complex. Use stubs for the remaining methods.
- Test a class at each stage of its development, particularly after adding a significant method.
- Using a fixed-size array can result in a full bag.
- Resizing an array makes it appear to change size. To do so, you allocate a new array, copy the entries from the original array to the new array, and use the original variable to reference the new array.
- Resizing an array enables you to implement collections whose contents are limited in number only by the size of the computer's memory.

Programming Tips

- When defining a class, implement and test a group of core methods. Begin with methods that add to a collection of objects and/or have involved implementations.
- A class should not return a reference to an array that is a private data field.
- Do not wait until you complete the implementation of an ADT before testing it. By writing stubs, which are incomplete definitions of required methods, you can begin testing early in the process.
- Even though you might have written a correct definition of a method, do not hesitate to revise it if you think of a better implementation.

- When increasing the size of an array, you copy its entries to a larger array. You should expand the array sufficiently to reduce the impact of the cost of copying. A common practice is to double the size of the array.
- A class implementing a single interface that declares the operations of an ADT should define the methods declared in the interface as its only public methods. However, the class can also define private methods and protected methods.

EXERCISES

1. Why are the methods `getIndexOf` and `removeEntry` in the class `ArrayBag` private instead of public?
2. Implement a method `replace` for the ADT bag that replaces and returns any object currently in a bag with a given object.
3. Revise the definition of the method `remove`, as given in Segment 2.24, so that it removes a random entry from a bag. Would this change affect any other method within the class `ArrayBag`?
4. Define a method `removeEvery` for the class `ArrayBag` that removes all occurrences of a given entry from a bag.
5. An instance of the class `ArrayBag` has a fixed size, whereas an instance of `ResizableArrayBag` does not. Give some examples of situations where a bag would be appropriate if its size is
   a. Fixed.
   b. Resizable.
6. Suppose that you wanted to define a class `PileOfBooks` that implements the interface described in Project 2 of the previous chapter. Would a bag be a reasonable collection to represent the pile of books? Explain.
7. Consider an instance `myBag` of the class `ResizableArrayBag`, as discussed in Segments 2.36 to 2.40. Suppose that the initial capacity of `myBag` is 10. What is the length of the array `bag` after
   a. Adding 145 entries to `myBag`?
   b. Adding an additional 20 entries to `myBag`?
8. Define a method at the client level that accepts as its argument an instance of the class `ArrayBag` and returns an instance of the class `ResizableArrayBag` that contains the same entries as the argument bag.
9. Suppose that a bag contains `Comparable` objects. Implement the following methods for the class `ArrayBag`:
   - The method `getMin` that returns the smallest object in a bag
   - The method `getMax` that returns the largest object in a bag
   - The method `removeMin` that removes and returns the smallest object in a bag
   - The method `removeMax` that removes and returns the largest object in a bag
10. Suppose that a bag contains `Comparable` objects. Define a method for the class `ArrayBag` that returns a new bag of items that are less than some given item. The header of the method could be as follows:

    ```
    public ArrayBag<T> getAllLessThan(Comparable<T> anObject)
    ```

    Make sure that your method does not affect the state of the original bag.
11. Define an `equals` method for the class `ArrayBag` that returns true when the contents of two bags are the same. Note that two equal bags contain the same number of entries, and each entry occurs in each bag the same number of times.

12. The class `ResizableArrayBag` has an array that can grow in size as objects are added to the bag. Revise the class so that its array also can shrink in size as objects are removed from the bag. Accomplishing this task will require two new private methods, as follows:

    - The first new method checks whether we should reduce the size of the array:

      ```
      private boolean isTooBig()
      ```

      This method returns true if the number of entries in the bag is less than half the size of the array and the size of the array is greater than 20.
    - The second new method creates a new array that is three quarters the size of the current array and then copies the objects in the bag to the new array:

      ```
      private void reduceArray()
      ```

    Implement each of these two methods, and then use them in the definitions of the two `remove` methods.

13. Consider the two private methods described in the previous exercise.

    a. The method `isTooBig` requires the size of the array to be greater than 20. What problem could occur if this requirement is dropped?

    b. The method `reduceArray` is not analogous to the method `ensureCapacity` in that it does not reduce the size of the array by one half. What problem could occur if the size of the array is reduced by one half instead of three quarters?

14. Define the method `union`, as described in Exercise 5 of the previous chapter, for the class `ResizableArrayBag`.

15. Define the method `intersection`, as described in Exercise 6 of the previous chapter, for the class `ResizableArrayBag`.

16. Define the method `difference`, as described in Exercise 7 of the previous chapter, for the class `ResizableArrayBag`.

PROJECTS

1. Define a class `ArraySet` that represents a set and implements the interface described in Project 1a of the previous chapter. Use the class `ResizableArrayBag` in your implementation. Then write a program that adequately demonstrates your implementation.

2. Repeat the previous project, but use a resizable array instead of the class `ResizableArrayBag`.

3. Define a class `PileOfBooks` that implements the interface described in Project 2 of the previous chapter. Use a resizable array in your implementation. Then write a program that adequately demonstrates your implementation.

4. Define a class `Ring` that represents a ring and implements the interface described in Project 3 of the previous chapter. Use a resizable array in your implementation. Then write a program that adequately demonstrates your implementation.

5. You can use either a set or a bag to create a spell checker. The set or bag serves as a dictionary and contains a collection of correctly spelled words. To see whether a word is spelled correctly, you see whether it is contained in the dictionary. Use this scheme to create a spell checker for the words in an external file. To simplify your task, restrict your dictionary to a manageable size.

6. Repeat the previous project to create a spell checker, but instead place the words whose spelling you want to check into a bag. The difference between the dictionary (the set or bag containing the correctly spelled words) and the bag of words to be checked is a bag of incorrectly spelled words.

Answers to Self-Test Questions

1. The students remain in consecutively numbered desks. You do not have to keep track of the locations of the empty desks.

2. Time is saved by not moving a student.

3. The student in the highest-numbered desk.

4. No. The two values are equal only when a bag is full.

5. If the client contained a statement such as

```
Object[] bagContents = myBag.toArray();
```

`myBag.getCurrentSize()` would be the number of entries in the array `bagContents`. With the proposed design, `bagContents.length` could be larger than the number of entries in the bag.

6. The statements set the first element of bag to `null`. The value of `numberOfEntries` does not change, so it is 5.

7.
```
public T[] toArray()
{
   return Arrays.copyOf(bag, bag.length);
} // end toArray
```

8. The bag `aBag` is empty. When `displayBag` is called, the statement

```
Object[] bagArray = aBag.toArray();
```

executes. When `toArray` is called, the statement

```
T[] result = (T[])new Object[numberOfEntries];
```

executes. Since `aBag` is empty, `numberOfEntries` is zero. Thus, the new array, `result`, is empty. The loop in `toArray` is skipped and the empty array is returned and assigned to `bagArray`. Since `bagArray.length` is zero, the loop in `displayBag` is skipped. The result of the call `displayBag(aBag)` is simply the line

```
The bag contains
```

9. Advantage: This definition is easier to write, so you are less likely to make a mistake.
Disadvantage: This definition takes more time to execute, if the bag contains more than one occurrence of `anEntry`. Note that the loop in the method `getFrequencyOf` cycles through all of the entries in the bag, whereas the loop in the method `contains`, as given in Segment 2.18, ends as soon as the desired entry is found.

10.
```
public void clear()
{
   while (remove() != null)
   {
   } // end while
} // end clear
```

11. Although the bag will appear empty to both the client and the other methods in `ArrayBag`, the references to the removed objects will remain in the array `bag`. Thus, the memory associated with these objects will not be deallocated.

12. By setting `bag[numberOfEntries]` to `null`, the method causes the memory assigned to the deleted entry to be recycled, unless another reference to that entry exists in the client.

13. An entry in the array `bag`, other than the last one, would be set to `null`. The remaining entries would no longer be in consecutive elements of the array. We could either rearrange the entries to get rid of the `null` entry or modify other methods to skip any `null` entry.

14. **a.** No. If `result` were `null`—and that is quite possible—a `NullPointerException` would occur.
b. Yes.

15. After locating `"B"` in the bag, the `remove` method replaces it with the last relevant entry in the array `bag`, which is `"C"`. It then replaces that last entry with `null`. Although we could define `remove` to result in either of the two other possibilities given in the question, both choices are inferior. For example, to get `"A", "A", "A", "C", null`, `remove` would shift the array elements, requiring more execution time. Leaving a gap in the array, such as `"A", "A", null, "A", "C"`, is easy for `remove` to do but complicates the logic of the remaining methods.

16.
```
assert ((where >= 0) && (where < numberOfEntries)) || (where == -1);
```

17.
```
private int getIndexOf(T anEntry)
{
   int where = -1;
   for (int index = 0; (where == -1) && (index < numberOfEntries); index++)
   {
      if (anEntry.equals(bag[index]))
         where = index;
   } // end for

   return where;
} // end getIndexOf
```

or

```
private int getIndexOf(T anEntry)
{
   int where = numberOfEntries - 1;
   while ((where > -1) && !anEntry.equals(bag[where]))
      where--;

   return where;
} // end getIndexOf
```

18.
```
text = Arrays.copyOf(text, text.length + 5);
```

or

```
String[] origText = text;
text = new String[text.length + 5];
System.arraycopy(origText, 0, text, 0, origText.length);
```

19.
```
text = Arrays.copyOf(text, size);
```

20.
```
/** Creates a bag containing the given array of entries.
    @param contents  an array of objects */
public ResizableArrayBag(T[] contents)
{
   bag = Arrays.copyOf(contents, contents.length);
   numberOfEntries = contents.length;
} // end constructor
```

21. A simple assignment statement would be a poor choice, since then the client could corrupt the bag's data by using the reference to the array that it passes to the constructor as an argument. Copying the argument array to the array `bag` is necessary to protect the integrity of the bag's data.

22. Advantage: You can access any array location directly if you know its index.
Disadvantages: The array has a fixed size, so you will either waste space or run out of room. Resizing the array avoids the latter disadvantage, but requires you to copy the contents of the original array to a larger array.