Chapter 7 Arrays and Array Lists

CHAPTER GOALS

- To become familiar with using arrays and array lists
- To learn about wrapper classes, auto-boxing, and the enhanced for loop
- To study common array algorithms
- To learn how to use two-dimensional arrays
- To understand when to choose array lists and arrays in your programs
- To implement partially filled arrays

T To understand the concept of regression testing

In order to process large quantities of data, you need to collect values in a data structure. The most commonly used data structures in Java are arrays and array lists. In this chapter, you will learn how to construct arrays and array lists, fill them with values, and access the stored values. We introduce the enhanced for loop, a convenient statement for processing all elements of a collection. You will see how to use the enhanced for loop, as well as ordinary loops, to implement common array algorithms. The chapter concludes with a technical section on copying array values.

287

288

7.1 Arrays

In many programs, you need to manipulate collections of related values. It would be impractical to use a sequence of variables such as `data1`, `data2`, `data3`, …, and so on. The array construct provides a better way of storing a collection of values.

An *array* is a sequence of values of the same type. For example, here is how you construct an array of 10 floating-point numbers:

```
new double[10]
```

The number of elements (here, 10) is called the length of the array.

An array is a sequence of values of the same type.

The `new` operator merely constructs the array. You will want to store a reference to the array in a variable so that you can access it later.

The type of an array variable is the element type, followed by `[]`. In this example, the type is `double[]`, because the element type is double. Here is the declaration of an array variable:

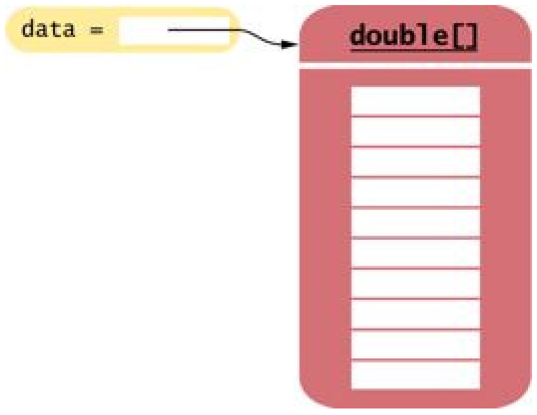
```
double[] data = new double[10];
```

That is, data is a reference to an array of floating-point numbers. It is initialized with an array of 10 numbers (see Figure 1).

You can also form arrays of objects, for example

```
BankAccount[] accounts = new BankAccount[10];
```

Figure 1



An Array Reference and an Array

When an array is first created, all values are initialized with 0 (for an array of numbers such as `int[]` or `double[]`), `false` (for a `boolean[]` array), or `null` (for an array of object references).

Each element in the array is specified by an integer index that is placed inside square brackets (`[]`). For example, the expression

```
data[4]
```

denotes the element of the data array with index 4.

You can store a value at a location with an assignment statement, such as the following.

```
data[2] = 29.95;
```

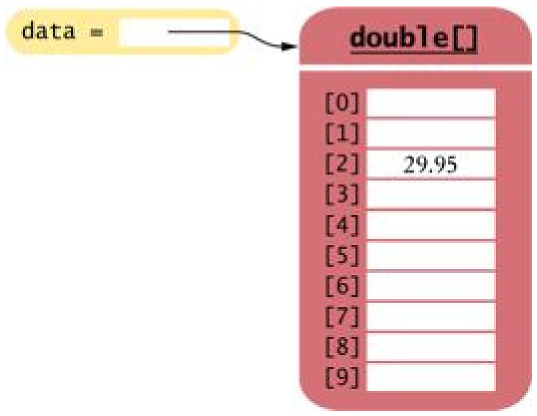
Now the position with index 2 of data is filled with the value 29.95 (see Figure 2).

You access array elements with an integer index, using the notation `a[i]`.

To read out the data value at index 2, simply use the expression `data[2]` as you would any variable of type `double`:

```
System.out.println("The value of this data item is "
      + data [2]);
```

Figure 2



Storing a Value in an Array

289

290

If you look closely at Figure 2, you will notice that the index values start at 0. That is,

`data[0]` is the first element

`data[1]` is the second element

`data[2]` is the third element

and so on. This convention can be a source of grief for the newcomer, so you should pay close attention to the index values. In particular, the *last* element in the array has an index *one less than* the array length. For example, data refers to an array with length 10. The last element is `data[9]`.

If you try to access an element that does not exist, then an exception is thrown. For example, the statement

```
data[10] = 29.95; // ERROR
```

is a bounds error.

Index values of an array range from 0 to `length - 1`. Accessing a nonexistent element results in a bounds error.

To avoid bounds errors, you will want to know how many elements are in an array. The `length field` returns the number of elements: `data.length` is the length of the `data` array. Note that there are no parentheses following `length`—it is an instance variable of the array object, not a method. However, you cannot assign a new value to this instance variable. In other words, `length` is a `final public` instance variable. This is quite an anomaly. Normally, Java programmers use a method to inquire about the properties of an object. You just have to remember to omit the parentheses in this case.

Use the `length` field to find the number of elements in an array.

The following code ensures that you only access the array when the index variable `i` is within the legal bounds:

```
if (0 <= i && i < data.length) data[i] = value;
```

Arrays suffer from a significant limitation: *their length is fixed*. If you start out with an array of 10 elements and later decide that you need to add additional elements, then you need to make a new array and copy all values of the existing array into the new array. We will discuss this process in detail in Section 7.7.

SYNTAX 7.1: Array Construction

`new` *typeName*[*length*]

Example:

```
new double[10]
```

Purpose:

To construct an array with a given number of elements

SYNTAX 7.2: Array Element Access

arrayReference[*index*]

Example:

```
data[2]
```

Purpose:

To access an element in an array

SELF CHECK

1. What elements does the data array contain after the following statements?

```
double[] data = new double[10];
for (int i = 0; i < data.length; i++) data[i] =
i * i;
```

2. What do the following program segments print? Or, if there is an error, describe the error and specify whether it is detected at compile-time or at run-time.

a.
```
double[] a = new double[10];
System.out.println(a[0]);
```

b.
```
double[] b = new double[10];
System.out.println(b[10]);
```

c.
```
double[] c;
System.out.println(c[0]);
```

COMMON ERROR 7.1: Bounds Errors

The most common array error is attempting to access a nonexistent position.

```
    double[] data = new double[10];
    data[10] = 29.95;
    // Error-only have elements with index values 0 ... 9
```

When the program runs, an out-of-bounds index generates an exception and terminates the program.

This is a great improvement over languages such as C and C++. With those languages there is no error message; instead, the program will quietly (or not so quietly) corrupt the memory location that is 10 elements away from the start of the array. Sometimes that corruption goes unnoticed, but at other times, the program will act flaky or die a horrible death many instructions later. These are serious problems that make C and C++ programs difficult to debug.

COMMON ERROR 7.2: Uninitialized Arrays

A common error is to allocate an array reference, but not an actual array.

```
    double[] data;
    data[0] = 29.95; // Error—data not initialized
```

Array variables work exactly like object variables—they are only references to the actual array. To construct the actual array, you must use the `new` operator:

```
    double[] data = new double[10];
```

ADVANCED TOPIC 7.1: Array Initialization

You can initialize an array by allocating it and then filling each entry:

```
int[] primes = new int[5];
primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
primes[3] = 7;
primes[4] = 11;
```

However, if you already know all the elements that you want to place in the array, there is an easier way. List all elements that you want to include in the array, enclosed in braces and separated by commas:

```
int[] primes = { 2, 3, 5, 7, 11 };
```

The Java compiler counts how many elements you want to place in the array, allocates an array of the correct size, and fills it with the elements that you specify.

If you want to construct an array and pass it on to a method that expects an array parameter, you can initialize an anonymous array as follows:

```
new int[] { 2, 3, 5, 7, 11 }
```

7.2 Array Lists

Arrays are a rather primitive construct. In this section, we introduce the `ArrayList` class that lets you collect objects, just like an array does. Array lists offer two significant conveniences:

292

293

The `ArrayList` class manages a sequence of objects.

- Array lists can grow and shrink as needed
- The `ArrayList` class supplies methods for many common tasks, such as inserting and removing elements

Let us define an array list of bank accounts and fill it with objects. (The `BankAccount` class has been enhanced from the version in Chapter 3. Each bank account has an account number.)

```
ArrayList<BankAccount> accounts = new
ArrayList<BankAccount>();
accounts.add(new BankAccount(1001));
accounts.add(new BankAccount(1015));
accounts.add(new BankAccount(1022));
```

The `ArrayList` class is a generic class: `ArrayList<T>` collects objects of type `T`.

The type `ArraList<BankAccount>` denotes an array list of bank accounts. The angle brackets around the `BankAccount` type tell you that BankAccount is a *type parameter*. You can replace `BankAccount` with any other class and get a different array list type. For that reason, `ArrayList` is called a *generic class*. You will learn more about generic classes in Chapter 17. For now, simply use an `ArrayList<T>` whenever you want to collect objects of type `T`. However, keep in mind that you cannot use primitive types as type parameters— there is no `ArrayList<int>` or `ArrayList<double>`.

When you construct an `ArrayList` object, it has size 0. You use the add method to add an object to the end of the array list. The size increases after each call to add. The `size` method yields the current size of the array list.

To get objects out of the array list, use the get method, not the `[ ]` operator. As with arrays, index values start at 0. For example, `accounts.get(2)` retrieves the account with index 2, the third element in the array list:

```
BankAccount anAccount = accounts.get(2);
```

As with arrays, it is an error to access a nonexistent element. The most common bounds error is to use the following:

```
int i = accounts.size();
anAccount = accounts.get(i); // Error
```

The last valid index is `accounts.size() - 1`.

To set an array list element to a new value, use the set method.

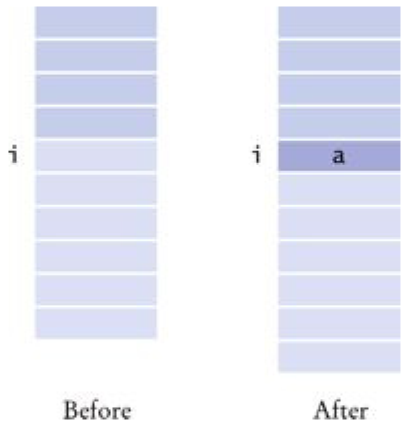
```
BankAccount anAccount = new BankAccount(1729);
accounts.set(2, anAccount);
```

This call sets position 2 of the `accounts` array list to `anAccount`, overwriting whatever value was there before.

The set method can only overwrite existing values. It is different from the add method, which adds a new object to the end of the array list.

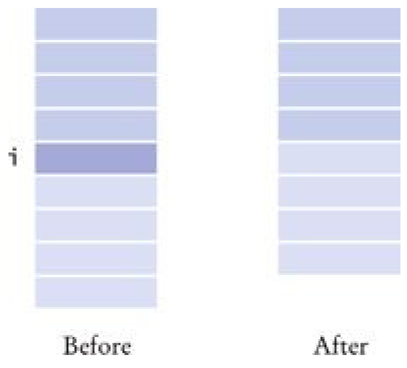
You can also insert an object in the middle of an array list. The call `accounts.add(i, a)` adds the object a at position `i` and moves all elements by one position, from the current element at position `i` to the last element in the array list.

Figure 3



Adding an Element in the Middle of an Array List.

Figure 4



Removing an Element from the Middle of an Array List

After each call to the add method, the size of the array list increases by 1 (see Figure 3).

Conversely, the call `accounts.remove(i)` removes the element at position `i`, moves all elements after the removed element down by one position, and reduces the size of the array list by 1 (see Figure 4).

The following program demonstrates the methods of the `ArrayList` class. Note that you import the generic class `java.util.ArrayList`, without the type parameter.

ch07/arraylist/ArrayListTester.java

```java
import java.util.ArrayList;

/**
   This program tests the ArrayList class.
*/
public class ArrayListTester
{
   public static void main(String[] args)
   {
      ArrayList<BankAccount> accounts
            = new ArrayList<BankAccount>();
      accounts.add(new BankAccount(1001));
      accounts.add(new BankAccount(1015));
```

294

295

```
        accounts.add(new BankAccount(1729));
        accounts.add(1, new BankAccount(1008));
        accounts.remove(0);

        System.out.println("Size: " +
accounts.size()) ;
        System.out.println("Expected: 3");
        BankAccount first = accounts.get(0);
        System.out.println("First account
number: "
              + first.getAccountNumber());
        System.out.println("Expected: 1008");
        BankAccount last =
accounts.get(accounts.size() - 1);
        System.out.println("Last account number:
"
              + last.getAccountNumber());
        System.out.println("Expected: 1729");
     }
  }
```

ch07/arraylist/ArrayListTester.java

```
  /**
     A bank account has a balance that can be
changed by
     deposits and withdrawals.
  */
  public class BankAccount
  {
     /**
        Constructs a bank account with a zero
balance.
        @param anAccountNumber the account
number for this account
    */
    public BankAccount(int anAccountNumber)
    {
        accountNumber = anAccountNumber;
        balance = 0;
    }

    /**
       Constructs a bank account with a given
balance.
```

```
        @param anAccountNumber the account number for this account
        @param initialBalance the initial balance
    */
    public BankAccount(int anAccountNumber, double initialBalance)
    {
        accountNumber = anAccountNumber;
        balance = initialBalance;
    }

```

```
    /**
        Gets the account number of this bank account.
        @return the account number
    */
    public int getAccountNumber()
    {
        return accountNumber;
    }

    /**
        Deposits money into the bank account.
        @param amount the amount to deposit
    */
    public void deposit(double amount)
    {
        double newBalance = balance + amount;
        balance = newBalance;
    }

    /**
        Withdraws money from the bank account.
        @param amount the amount to withdraw
    */
    public void withdraw(double amount)
    {
       double newBalance = balance - amount;
        balance = newBalance;
    }

    /**
        Gets the current balance of the bank account.
        @return the current balance
    */
    public double getBalance()
    {
        return balance;
    }

    private int accountNumber;
    private double balance;
}
```

Output

```
Size: 3
Expected: 3
First account number: 1008
Expected: 1008
Last account number: 1729
Expected: 1729
```

SELF CHECK

3. How do you construct an array of 10 strings? An array list of strings?

4. What is the content of names after the following statements?

```
ArrayList<String> names =  new
ArrayList<String>();
names.add("A");
names.add(0, "B");
names.add("C");
names.remove(1);
```

COMMON ERROR 7.3: Length and Size

Unfortunately, the Java syntax for determining the number of elements in an array, an array list, and a string is not at all consistent. It is a common error to confuse these. You just have to remember the correct syntax for every data type.

Data Type	Number of Elements
Array	`a.length`
Array list	`a.size()`
String	`a.length()`

QUALITY TIP 7.1: Prefer Parameterized Array Lists

Parameterized array lists, such as `ArrayList<BankAccount>`, were introduced to the Java language in 2004. Versions of Java prior to version 5.0 had only an untyped class `ArrayList`. The untyped array list can hold elements of any class. (Technically, it holds elements of type Object, the "lowest common denominator" of all Java classes.) Whenever you retrieve an element from an untyped array list, the compiler requires you to use a cast:

```
ArrayList accounts = new ArrayList();    // Untyped
ArrayList
accounts.add(new BankAccount(1729));     // OK—can add
any object
BankAccount a = (BankAccount) a.get(0); // Need cast
```

The cast is needed because the compiler does not keep track of the objects that were inserted into the array list, and the array list `get` method has return type `Object`.

Untyped array lists are still a part of the Java language—after all, we want to continue to use programs that were written before 2004. But you should not use them for new code. The casts are tedious and also a bit error-prone. If you apply the wrong cast, the compiler cannot detect your mistake. Instead, your program will throw an exception.

297

298

7.3 Wrappers and Auto-Boxing

Because numbers are not objects in Java, you cannot directly insert them into array lists. For example, you cannot form an `ArrayList<double>`. To store sequences of numbers in an array list, you must turn them into objects by using wrapper classes.

To treat primitive type values as objects, you must use wrapper classes.

There are wrapper classes for all eight primitive types:

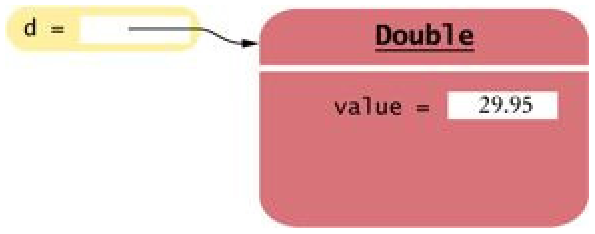
Primitive Type	Wrapper Class
`byte`	`Byte`
`boolean`	`Boolean`
`char`	`Character`
`double`	`Double`
`float`	`Float`
`int`	`Integer`
`long`	`Long`
`short`	`Short`

Note that the wrapper class names start with uppercase letters, and that two of them differ from the names of the corresponding primitive type: `Integer` and `Character`.

Each wrapper class object contains a value of the corresponding primitive type. For example, an object of the class `Double` contains a value of type `double` (see Figure 5).

Wrapper objects can be used anywhere that objects are required instead of primitive type values. For example, you can collect a sequence of floating-point numbers in an `ArrayList<Double>`.

Figure 5



An Object of a Wrapper Class

Starting with Java version 5.0, conversion between primitive types and the corresponding wrapper classes is automatic. This process is called *auto-boxing* (even though *auto-wrapping* would have been more consistent).

For example, if you assign a number to a `Double` object, the number is automatically “put into a box”, namely a wrapper object.

```
Double d = 29.95; // auto-boxing; same as Double d = new
Double(29.95);
```

If you use an older version of Java, you need to provide the constructor yourself.

Conversely, starting with Java version 5.0, wrapper objects are automatically "unboxed" to primitive types.

```
double x = d; // auto-unboxing; same as double x =
d.doubleValue();
```

With older versions, you need to call a method such as `doubleValue`, `intValue`, or `booleanValue` for unboxing.

Auto-boxing even works inside arithmetic expressions. For example, the statement

```
Double e = d + 1;
```

is perfectly legal. It means:

- Auto-unbox `d` into a `double`
- Add 1
- Auto-box the result into a new `Double`
- Store a reference to the newly created wrapper object in `e`

If you use Java version 5.0 or higher, array lists of numbers are straightforward. Simply remember to use the wrapper type when you declare the array list, and then rely on auto-boxing.

```
ArrayList<Double> data = new ArrayList<Double>();
data.add(29.95);
double x = data.get(0);
```

With older versions of Java, using wrapper classes to store numbers in an array list is a considerable hassle because you must manually box and unbox the numbers.

No matter which Java version you use, you should know that storing wrapped numbers is quite inefficient. The use of wrappers is acceptable for short array lists, but you should use arrays for long sequences of numbers or characters.

SELF CHECK

5. What is the difference between the Types `double` and `Double`?

6. Suppose data is an `ArrayList<Double>` of size > 0. How do you increment the element with index 0?

7.4 The Enhanced for Loop

Java version 5.0 introduces a very convenient shortcut for a common loop type. Often, you need to iterate through a sequence of elements—such as the elements of an array or array list. The enhanced `for` loop makes this process particularly easy to program.

The enhanced for loop traverses all elements of a collection.

Suppose you want to total up all data values in an array data. Here is how you use the enhanced `for` loop to carry out that task.

```
double[] data = . . .;
double sum = 0;
for (double e : data)
{
   sum = sum + e;
}
```

The loop body is executed for each element in the array `data`. At the beginning of each loop iteration, the next element is assigned to the variable `e`. Then the loop body is executed. You should read this loop as “for each `e` in data”.

You may wonder why Java doesn't let you write “`for each (e in data)`”. Unquestionably, this would have been neater, and the Java language designers seriously considered this. However, the “for each” construct was added to Java several years after its initial release. Had new keywords each and `in` been added to the language, then older programs that happened to use those identifiers as variable or method names (such as `System.in`) would no longer have compiled correctly.

You don't have to use the “for each” construct to loop through all elements in an array. You can implement the same loop with a straightforward `for` loop and an explicit index variable:

```
double[] data = . . .;
double sum = 0;
for (int i = 0; i < data.length; i++)
{
   double e = data[i];
   sum = sum + e;
}
```

Note an important difference between the “for each” loop and the ordinary for loop. In the “for each” loop, the *element variable* `e` is assigned values `data[0]`, `data[1]`, and so on. In the ordinary `for` loop, the *index variable* `i` is assigned values 0, 1, and so on.

You can also use the enhanced for loop to visit all elements of an array list. For example, the following loop computes the total value of all accounts:

```
ArrayList<BankAccount> accounts = . . . ;
double sum = 0;
for (BankAccount a : accounts)
{
   sum = sum + a.getBalance();
}
```

This loop is equivalent to the following ordinary `for` loop:

```
double sum = 0;
for (int i = 0; i < accounts.size(); i++)
{
   BankAccount a = accounts.get(i);
   sum = sum + a.getBalance();
}
```

The “for each” loop has a very specific purpose: traversing the elements of a collection, from the beginning to the end. Sometimes you don't want to start at the beginning, or you may need to traverse the collection backwards. In those situations, do not hesitate to use an ordinary `for` loop.

SYNTAX 7.3: The "for each" Loop

`for` (*Type variable : collection*) *statement*

Example:

```
for (double e : data)
   sum = sum + e;
```

Purpose:

To execute a loop for each element in the collection. In each iteration, the variable is assigned the next element of the collection. Then the statement is executed.

SELF CHECK

7. Write a "for each" loop that prints all elements in the `array data`.

8. Why is the "for each" loop not an appropriate shortcut for the following ordinary `for` loop?

```
for (int i = 0; i < data.length; i++) data[i] =
i * i;
```

7.5 Simple Array Algorithms

7.5.1 Counting Matches

To count values in an array list, check all elements and count the matches until you reach the end of the array list.

Suppose you want to find how many accounts of a certain type you have. Then you must go through the entire collection and increment a counter each time you find a match. Here we count the number of accounts whose balance is at least as much as a given threshold:

```
public class Bank
{
```
301

302
```
   public int count(double atLeast)
```

```
      {
         int matches = 0;
         for (BankAccount a : accounts)
         {
            if (a.getBalance() >= atLeast) matches++;
               // Found a match
         }
         return matches;
      }
      . . .
      private ArrayList<BankAccount> accounts;
   }
```

7.5.2 Finding a Value

Suppose you want to know whether there is a bank account with a particular account number in your bank. Simply inspect each element until you find a match or reach the end of the array list. Note that the loop might fail to find an answer, namely if none of the accounts match. This search process is called a linear search through the array list.

To find a value in an array list, check all elements until you have found a match.

```
public class Bank
{
   public BankAccount find(int accountNumber)
   {
      for (BankAccount a : accounts)
      {
         if (a.getAccountNumber() == accountNumber)//
Found a match
            return a;
      }
      return null; // No match in the entire array list
   }
   . . .
}
```

Note that the method returns `null` if no match is found.

7.5.3 Finding the Maximum or Minimum

Suppose you want to find the account with the largest balance in the bank. Keep a candidate for the maximum. If you find an element with a larger value, then replace the candidate with that value. When you have reached the end of the array list, you have found the maximum.

To compute the maximum or minimum value of an array list, initialize a candidate with the starting element. Then compare the candidate with the remaining elements and update it if you find a larger or smaller value.

There is just one problem. When you visit the beginning of the array, you don't yet have a candidate for the maximum. One way to overcome that is to set the candidate to the starting element of the array and start the comparison with the next element. 302

303

```
BankAccount largestYet = accounts.get(0);
for (int i = 1; i < accounts.size(); i++)
{
   BankAccount a = accounts.get(i);
   if (a.getBalance() > largestYet.getBalance())
      largestYet = a;
}
return largestYet;
```

Now we use an explicit `for` loop because the loop no longer visits all elements—it skips the starting element.

Of course, this approach works only if there is at least one element in the array list. It doesn't make a lot of sense to ask for the largest element of an empty collection. We can return `null` in that case:

```
if (accounts.size() == 0) return null;
BankAccount largestYet = accounts.get(0);
. . .
```

See Exercises R7.5 and R7.6 for slight modifications to this algorithm.

To compute the minimum of a data set, keep a candidate for the minimum and replace it whenever you encounter a *smaller* value. At the end of the array list, you have found the minimum.

The following sample program implements a `Bank` class that stores an array list of bank accounts. The methods of the `Bank` class use the algorithms that we have discussed in this section.

ch07/bank/Bank.java

```
import java.util.ArrayList;

/**
   This bank contains a collection of bank accounts.
*/
public class Bank
{
   /**
      Constructs a bank with no bank accounts.
   */
   public Bank()
   {
      accounts = new ArrayList<BankAccount>();
   }

   /**
      Adds an account to this bank.
      @param a the account to add
   */
   public void addAccount(BankAccount a)
   {
      accounts.add(a);
   }

   /**
      Gets the sum of the balances of all accounts in this bank.
      @return the sum of the balances
   */
   public double getTotalBalance()
   {
      double total = 0;
```

```java
        for (BankAccount a : accounts)
        {
            total = total + a.getBalance();
        }
        return total;
    }

    /**
        Counts the number of bank accounts whose balance is at
        least a given value.
        @param atLeast the balance required to count an account
        @return the number of accounts having at least the given balance
    */
    public int count(double atLeast)
    {
        int matches = 0;
        for (BankAccount a : accounts)
        {
            if (a.getBalance() >= atLeast) matches++;// Found a match
        }
         return matches;
    }

    /**
        Finds a bank account with a given number.
        @param accountNumber the number to find
        @return the account with the given number, or null if there
        is no such account
    */
    public BankAccount find(int accountNumber)
    {
        for (BankAccount a : accounts)
        {
            if (a.getAccountNumber() == accountNumber)// Found a match
                return a;
        }
        return null;// No match in the entire array list
    }

    /**
        Gets the bank account with the largest balance.
        @return the account with the largest balance, or null if the
        bank has no accounts
    */
    public BankAccount getMaximum()
    {
         if (accounts.size() == 0) return null;
         BankAccount largestYet = accounts.get(0);
         for (int i = 1; i < accounts.size(); i++)
        {
           BankAccount a = accounts.get(i);
           if (a.getBalance() > largestYet.getBalance())
              largestYet = a;
        }
        return largestYet;
    }

    private ArrayList<BankAccount> accounts;
 }
```

ch07/bank/BankTester.java

```
/**
   This program tests the Bank class.
*/
public class BankTester
{
   public static void main(String[] args)
   {
      Bank firstBankOfJava = new Bank();
      firstBankOfJava.addAccount(new BankAccount(1001, 20000));
      firstBankOfJava.addAccount(new BankAccount(1015, 10000));
      firstBankOfJava.addAccount(new BankAccount(1729, 15000));

      double threshold = 15000;
```

```
          int c = firstBankOfJava.count(threshold);
          System.out.println("Count: " + c);
          System.out.println("Expected: 2");

          int accountNumber = 1015;
          BankAccount a = firstBankOfJava.find(accountNumber);
          if (a == null)
             System.out.println("No matching account");
          else
             System.out.println("Balance of matching account: "
                   + a.getBalance());
          System.out.println("Expected: 10000");

          BankAccount max = firstBankOfJava.getMaximum();
          System.out.println("Account with largest balance: "
                + max.getAccountNumber());
          System.out.println("Expected: 1001");
     }
  }
```

Output

```
Count: 2
Expected: 2
Balance of matching account: 10000.0
Expected: 10000
Account with largest balance: 1001
Expected: 1001
```

SELF CHECK

9. What does the `find` method do if there are two bank accounts with a matching account number?

10. Would it be possible to use a “for each” loop in the `getMaximum` method?

7.6 Two-Dimensional Arrays

Arrays and array lists can store linear sequences. Occasionally you want to store collections that have a two-dimensional layout. The traditional example is the tic-tac-toe board (see Figure 6).

Two-dimensional arrays form a tabular, two-dimensional arrangement. You access elements with an index pair `a[i][j]`.

Such an arrangement, consisting of rows and columns of values, is called a two-dimensional array or matrix. When constructing a two-dimensional array, you specify how many rows and columns you need. In this case, ask for 3 rows and 3 columns:

```
final int ROWS = 3;
final int COLUMNS = 3;
String[][] board = new String [ROWS][COLUMNS];
```

This yields a two-dimensional array with 9 elements

```
board[0][0] board[0][1] board[0][2]
board[1][0] board[1][1] board[1][2]
board[2][0] board[2][1] board[2][2]
```

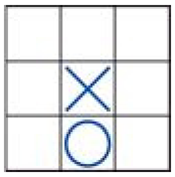
To access a particular element, specify two subscripts in separate brackets:

```
board[i][j] = "x";
```

When filling or searching a two-dimensional array, it is common to use two nested loops. For example, this pair of loops sets all elements in the array to spaces.

```
for (int i = 0; i < ROWS; i++)
   for (int j = 0; j < COLUMNS; j++)
      board[i][j] = " ";
```

Figure 6



A Tic-Tac-Toe Board

Here is a class and a test program for playing tic-tac-toe. This class does not check whether a player has won the game. That is left as the proverbial "exercise for the reader"—see Exercise P7.10.

ch07/twodim/TicTacToe.java

```
/**
   A 3 x 3 tic-tac-toe board.
*/
public class TicTacToe
{
   /**
      Constructs an empty board.
   */
   public TicTacToe()
   {
      board = new String[ROWS][COLUMNS];
      //Fill with spaces
      for (int i = 0; i < ROWS; i++)
         for (int j = 0; j < COLUMNS; j++)
            board[i][j] = " ";
   }

   /**
      Sets a field in the board. The field
must be unoccupied.
      @param i the row index
      @param j the column index
      @param player the player ("x" or "o")
   */
   public void set(int i, int j, String player)
   {
      if (board[i][j].equals(" "))
         board[i][j] = player;
   }

     /**
      Creates a string representation of the board, such as
      |x o|
      | x |
      |  o|.
      @return the string representation
   */
   public String toString()
   {
      String r = "";
      for (int i = 0; i < ROWS; i++)
      {
         r = r + "|";
         for (int j = 0; j < COLUMNS; j++)
            r = r + board[i][j];
         r = r + "|\n";
      }
      return r;
 }
```

```

 private String[][] board;
 private static final int ROWS = 3;
 private static final int COLUMNS = 3;
}
```

ch07/twodim/TicTacToeRunner.java

```
import java.util. Scanner;

/**
   This program runs a TicTacToe game. It prompts the
   user to set positions on the board and prints out the
   result.
*/
public class TicTacToeRunner
{
 public static void main(String[] args)
 {
      Scanner in = new Scanner(System.in);
      String player = "x";
      TicTacToe game = new TicTacToe();
      boolean done = false;
      while (!done)
      {
         System.out.print(game.toString());
         System.out.print(
```

308

309

```
               "Row for " + player + " (-1 to exit): ");
         int row = in.nextInt();
         if (row < 0) done = true;
         else
         {
            System.out.print("Column for " + player + ": ");
            int column = in.nextInt();
            game.set(row, column, player);
            if (player.equals("x"))
               player = "o";
            else
               player = "x";
         }
      }
   }
}
```

Output

```
|   |
|   |
|   |
Row for x (-1 to exit): 1
Column for x: 2
|   |
|  x|
|   |
Row for o (-1 to exit): 0
Column for o: 0
|o  |
|  x|
|   |
Row for x (-1 to exit): -1
```

SELF CHECK

11. How do you declare and initialize a 4-by-4 array of integers?

12. How do you count the number of spaces in the tic-tac-toe board?

HOW TO 7.1: Working with Array Lists and Arrays

Step 1 Pick the appropriate data structure.

As a rule of thumb, your first choice should be an array list. Use an array if you collect numbers (or other primitive type values) and efficiency is an issue, or if you need a two-dimensional array.

Step 2 Construct the array list or array and save a reference in a variable.

For both array lists and arrays, you need to specify the element type. For an array, you also need to specify the length.

```
ArrayList<BankAccount> accounts = new
ArrayList<BankAccount>();
double[] balances = new double[n];
```

Step 3 Add elements.

For an array list, simply call the `add` method. Each call adds an element at the end.

```
accounts.add(new BankAccount(1008));
accounts.add(new BankAccount(1729));
```

For an array, you use index values to access the elements.

```
balance[0] = 29.95;
balance[1] = 1000;
```

Step 4 Process elements.

The most common processing pattern involves visiting all elements in the collection. Use the "for each" loop for this purpose:

```
for (BankAccount a : accounts)
```

```
   Do something with a
```

If you don't need to look at all of the elements, use an ordinary loop instead. For example, to skip the initial element, you can use this loop.

```
for (int i = 1; i < accounts.size(); i++)
{
   BankAccount a = accounts.get(i);
      Do something with a
}
```

For arrays, you use `.length` instead of `.size()` and `[i]` instead of `.get(i)`.

ADVANCED TOPIC 7.2: Two-Dimensional Arrays with Variable Row Lengths

When you declare a two-dimensional array with the command

```
int[][] a = new int[5][5];
```

then you get a 5-by-5 matrix that can store 25 elements:

```
a[0][0] a[0][1] a[0][2] a[0][3] a[0][4]
a[1][0] a[1][1] a[1][2] a[1][3] a[1][4]
a[2][0] a[2][1] a[2][2] a[2][3] a[2][4]
a[3][0] a[3][1] a[3][2] a[3][3] a[3][4]
a[4][0] a[4][1] a[4][2] a[4][3] a[4][4]
```

In this matrix, all rows have the same length. In Java it is possible to declare arrays in which the row length varies. For example, you can store an array that has a triangular shape, such as:

```
b[0][0]
b[1][0] b[1][1]
b[2][0] b[2][1] b[2][2]
b[3][0] b[3][1] b[3][2] b[3][3]
b[4][0] b[4][1] b[4][2] b[4][3] b[4][4]
```

To allocate such an array, you must work harder. First, you allocate space to hold five rows. Indicate that you will manually set each row by leaving the second array index empty:

```
int[][] b = new int[5][];
```

Then allocate each row separately.

```
for (int i = 0; i < b.length; i++)
   b[i] = new int[i + 1];
```

You can access each array element as `b[i][j]`, but be careful that `j` is less than `b[i].length`.

Naturally, such “ragged” arrays are not very common.

ADVANCED TOPIC 7.3: Multidimensional Arrays

You can declare arrays with more than two dimensions. For example, here is a three-dimensional array:

```
int[][][] rubiksCube = new int[3][3][3];
```

Each array element is specified by three index values,

```
rubiksCube[i][j][k]
```

However, these arrays are quite rare, particularly in object-oriented programs, and we will not consider them further.

310

311

7.7 Copying Arrays

Array variables work just like object variables—they hold a reference to the actual array. If you copy the reference, you get another reference to the same array (see Figure 7):

An array variable stores a reference to the array. Copying the variable yields a second reference to the same array.

```
double[] data = new double[10];
. . .// Fill array
double[] prices = data;
```

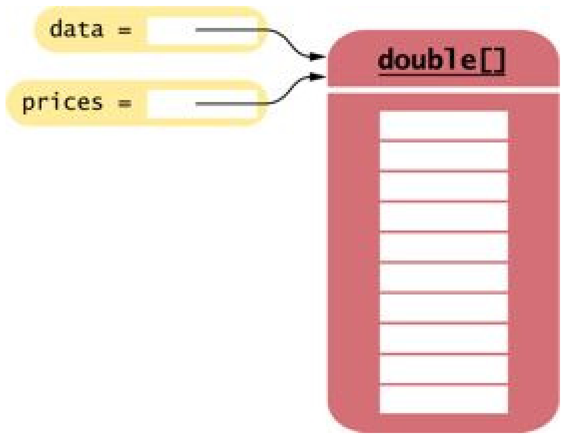
If you want to make a true copy of an array, call the `clone` method (see Figure 8).

Use the `clone` method to copy the elements of an array.

```
double[] prices = (double[]) data.clone();
```

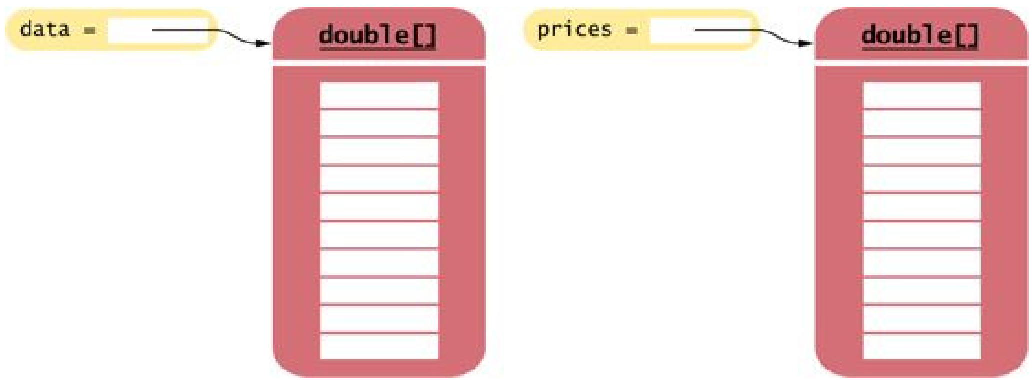
The `clone` method (which we will more closely study in Chapter 10) has the return type `Object`. You need to cast the return value of the clone method to the appropriate array type such as `double[]`.

Figure 7



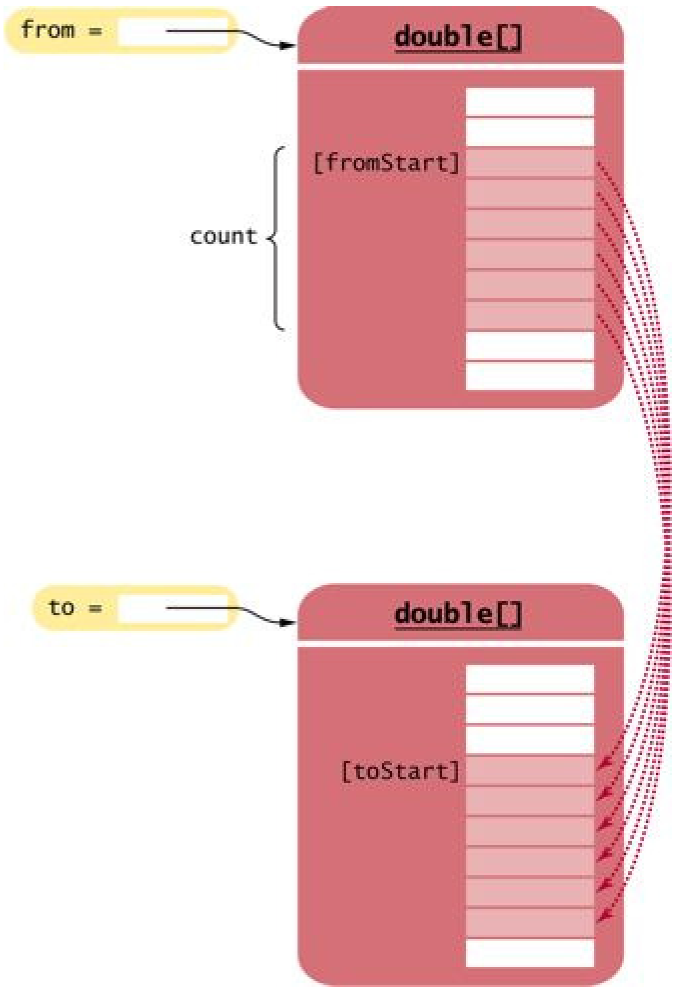
Two References to the Same Array

Figure 8



Cloning an Array

Figure 9



The `System.arraycopy` Method

Occasionally, you need to copy elements from one array into another array. You can use the static `System.arraycopy` method for that purpose (see Figure 9):

Use the `System.arraycopy` method to copy elements from one array to another.

```
System.arraycopy(from, fromStart, to, toStart,
count);
```

One use for the `System.arraycopy` method is to add or remove elements in the middle of an array. To add a new element at position `i` into data, first move all elements from `i` onward one position up. Then insert the new value.

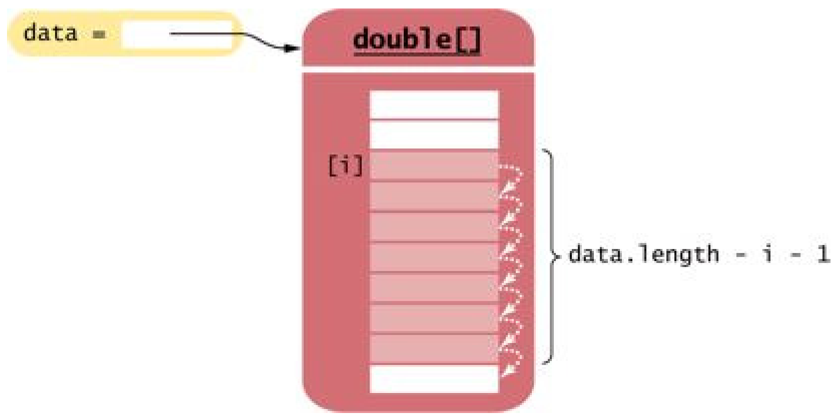
```
System.arraycopy(data, i, data, i + 1, data.length -
i - 1); data[i] = x;
```

Note that the last element in the array is lost (see Figure 10).

To remove the element at position `i`, copy the elements above the position downward (see Figure 11).

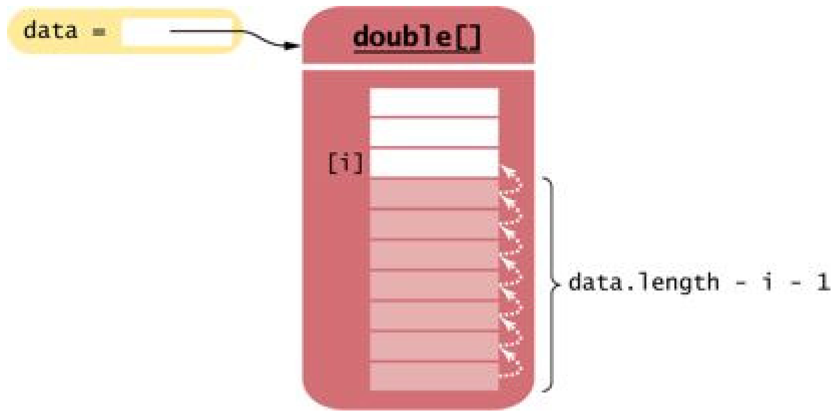
```
System.arraycopy(data, i + 1, data, i, data.length -
i - 1);
```

Figure 10



Inserting a New Element into an Array

Figure 11



Removing an Element from an Array

Another use for `System.arraycopy` is to grow an array that has run out of space. Follow these steps:

- Create a new, larger array.

```
double[] newData = new double[2 * data.length];
```

- Copy all elements into the new array

```
System.arraycopy(data, 0, newData, 0,
data.length);
```

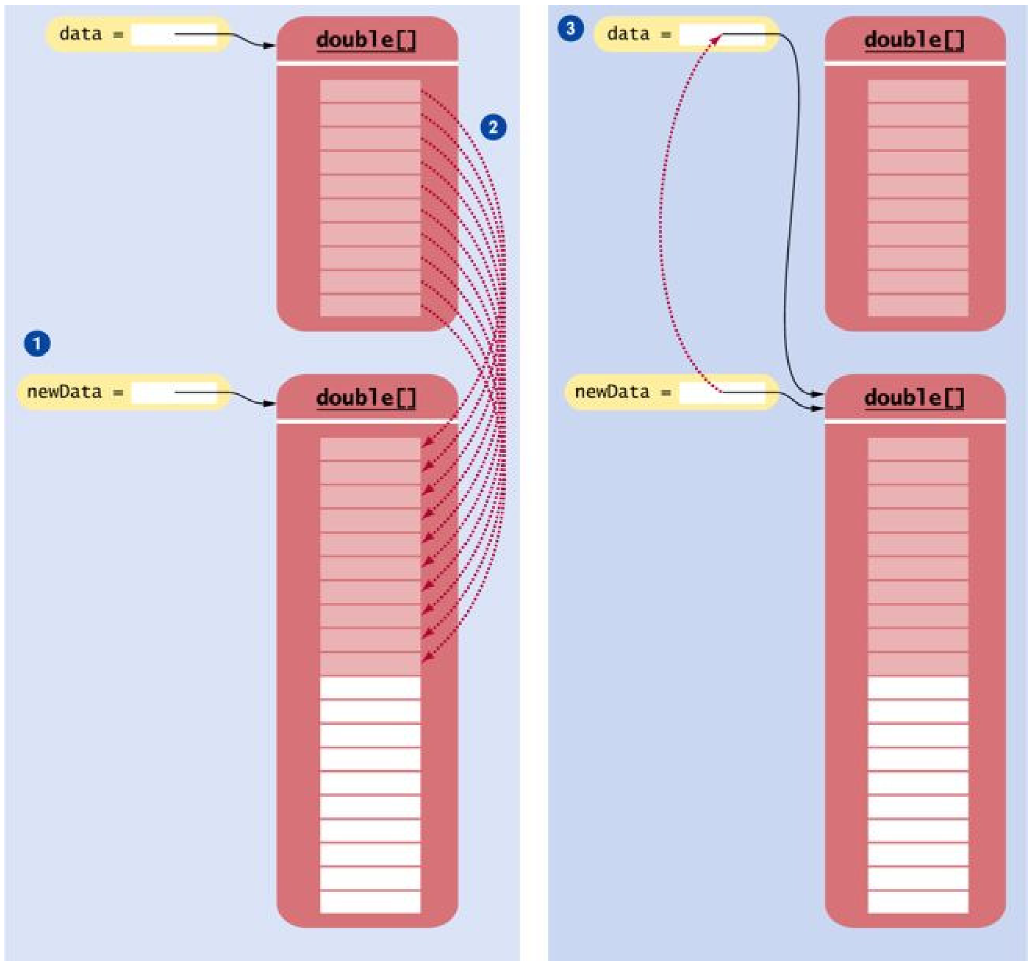
- Store the reference to the new array in the array variable.

```
data = newData;
```

Figure 12 shows the process.

313

Figure 12



Growing an Array

SELF CHECK

13. How do you add or remove elements in the middle of an array list?

14. Why do we double the length of the array when it has run out of space rather than increasing it by one element?

COMMON ERROR 7.4: Underestimating the Size of a Data Set

Programmers commonly underestimate the amount of input data that a user will pour into an unsuspecting program. The most common problem caused by underestimating the amount of input data results from the use of fixed-sized arrays. Suppose you write a program to search for text in a file. You store each line in a string, and keep an array of strings. How big do you make the array? Surely nobody is going to challenge your program with an input that is more than 100 lines. Really? A smart grader can easily feed in the entire text of *Alice in Wonderland* or *War and Peace* (which are available on the Internet). All of a sudden, your program has to deal with tens or hundreds of thousands of lines. What will it do? Will it handle the input? Will it politely reject the excess input? Will it crash and burn?

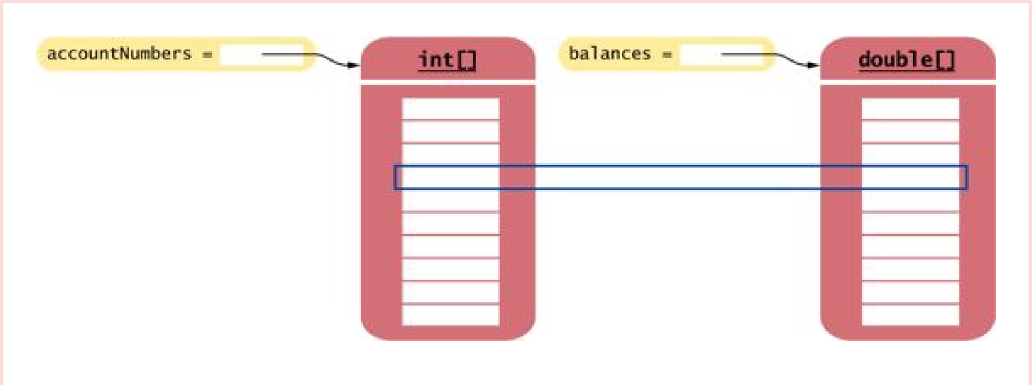
A famous article [1] analyzed how several UNIX programs reacted when they were fed large or random data sets. Sadly, about a quarter didn't do well at all, crashing or hanging without a reasonable error message. For example, in some versions of UNIX the tape backup program tar cannot handle file names that are longer than 100 characters, which is a pretty unreasonable limitation. Many of these shortcomings are caused by features of the C language that, unlike Java, make it difficult to store strings of arbitrary size.

QUALITY TIP 7.2: Make Parallel Arrays into Arrays of Objects

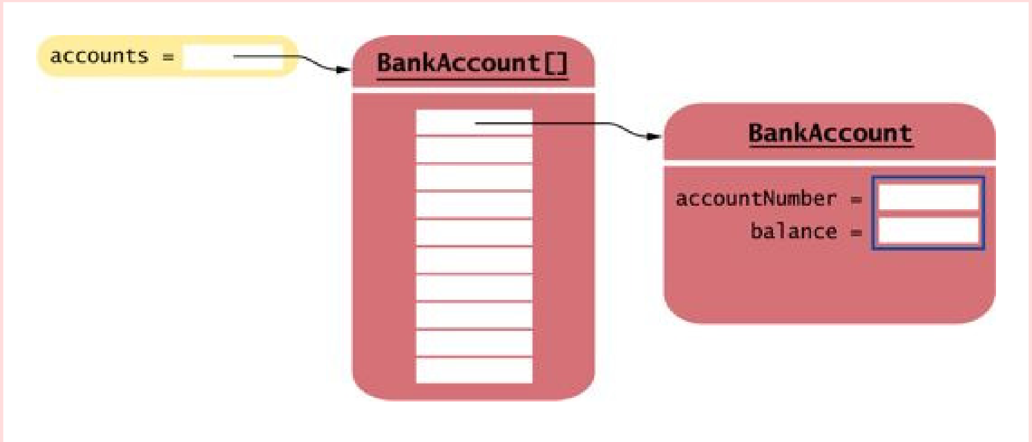
Programmers who are familiar with arrays, but unfamiliar with object-oriented programming, sometimes distribute information across separate arrays. Here is a typical example. A program needs to manage bank data, consisting of account numbers and balances. Don't store the account numbers and balances in separate arrays.

```
// Don’t do this
int[] accountNumbers;
double[] balances;
```

Arrays such as these are called parallel arrays (see Avoid Parallel Arrays). The `i`th slice (`accountNumbers[i]` and `balances[i]`) contains data that need to be processed together.



Avoid Parallel Arrays



Reorganizing Parallel Arrays into an Array of Objects

Avoid parallel arrays by changing them into arrays of objects.

If you find yourself using two arrays that have the same length, ask yourself whether you couldn't replace them with a single array of a class type. Look at a slice and find the concept that it represents. Then make the concept into a class. In our example each slice contains an account number and a balance, describing a bank account. Therefore, it is an easy matter to use a single array of objects

```
BankAccount[] accounts;
```

(See figure above.) Or, even better, use an `ArrayList<BankAccount>`.

Why is this beneficial? Think ahead. Maybe your program will change and you will need to store the owner of the bank account as well. It is a simple matter to update the `BankAccount` class. It may well be quite complicated to add a new array and make sure that all methods that accessed the original two arrays now also correctly access the third one.

ADVANCED TOPIC 7.4: Partially Filled Arrays

Suppose you write a program that reads a sequence of numbers into an array. How many numbers will the user enter? You can't very well ask the user to count the items before entering them—that is just the kind of work that the user expects the computer to do. Unfortunately, you now run into a problem. You need to set the size of the array before you know how many elements you need. Once the array size is set, it cannot be changed.

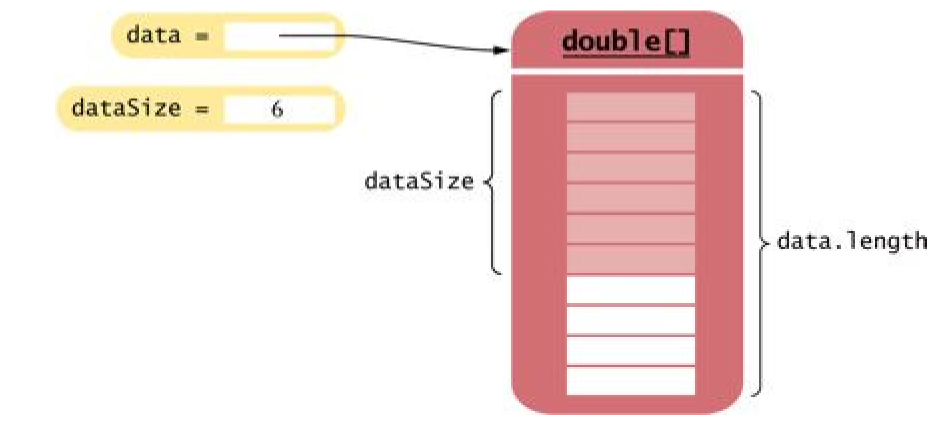
To solve this problem, make an array that is guaranteed to be larger than the largest possible number of entries, and partially fill it. For example, you can decide that the user will never enter more than 100 data values. Then allocate an array of size 100:

```
final int DATA_LENGTH = 100;
double[] data = new double[DATA_LENGTH];
```

Then keep a companion variable that tells how many elements in the array are actually used. It is an excellent idea always to name this companion variable by adding the suffix `Size` to the name of the array.

```
int dataSize = 0;
```

316



A Partially Filled Array

Now `data.length` is the capacity of the array data, and `dataSize` is the current size of the array (see A Partially Filled Array). Keep adding elements into the array, incrementing the `dataSize` variable each time.

```
    data[dataSize] = x;
    dataSize++;
```

This way, `dataSize` always contains the correct element count. When you run out of space, make a new array and copy the elements into it, as described in the preceding section.

Array lists use this technique behind the scenes. An array list contains an array of objects. When the array runs out of space, the array list allocates a larger array and copies the data. However, all of this happens inside the array list methods, so you never need to think about it.

ADVANCED TOPIC 7.5: Methods with a Variable Number of Parameters

Starting with Java version 5.0, it is possible to declare methods that receive a variable number of parameters. For example, we can modify the add method of the `DataSet` class of Chapter 6 so that one can add any number of values:

```
    data.add(1, 3, 7);
    data.add(4);
    data.add(); // OK but not useful
```

The modified add method must be declared as

```
    public void add(double... xs)
```

The ... symbol indicates that the method can receive any number of `double` values. The `xs` parameter is actually a `double[]` array that contains all values that were passed to the method. The method implementation traverses the parameter array and processes the values:

```
    for (x : xs)
    {
       sum = sum + x;
    }
```

RANDOM FACT 7.1: An Early Internet Worm

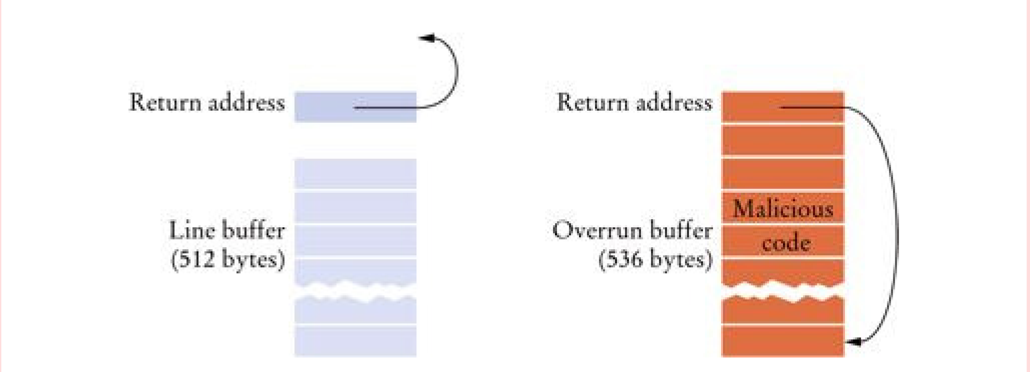
In November 1988, a graduate student at Cornell University launched a virus program that infected about 6,000 computers connected to the Internet across the United States. Tens of thousands of computer users were unable to read their e-mail or otherwise use their computers. All major universities and many high-tech companies were affected. (The Internet was much smaller then than it is now.)

The particular kind of virus used in this attack is called a worm. The virus program crawled from one computer on the Internet to the next. The entire program is quite complex; its major parts are explained in [2]. However, one of the methods used in the attack is of interest here. The worm would attempt to connect to `finger`, a program in the UNIX operating system for finding information on a user who has an account on a particular computer on the network. Like many programs in UNIX, `finger` was written in the C language. C does not have array lists, only arrays, and when you construct an array in C, as in Java, you have to make up your mind how many elements you need. To store the user name to be looked up (say, `walters@cs.sjsu.edu`), the finger program allocated an array of 512 characters, under the assumption that nobody would ever provide such a long input. Unfortunately, C, unlike Java, does not check that an array index is less than the length of the array. If you write into an array, using an index that is too large, you simply overwrite memory locations that belong to some other objects. In some versions of the finger program, the programmer had been lazy and had not checked whether the array holding the input characters was large enough to hold the input. So the worm program purposefully filled the 512-character array with 536 bytes. The excess 24 bytes would overwrite a return address, which the attacker knew was stored just after the line buffer. When that function was finished, it didn't return to its caller but to code supplied by the worm (see A "Buffer Overrun" Attack). That code ran under the same super-user privileges as finger, allowing the worm to gain entry into the remote system.

Had the programmer who wrote finger been more conscientious, this particular attack would not be possible. In C++ and C, all programmers must be especially careful not to overrun array boundaries.

One may well wonder what would possess a skilled programmer to spend many weeks or months to plan the antisocial act of breaking into thousands of computers and disabling them. It appears that the break-in was fully intended by the author, but the disabling of the computers was a side effect of continuous reinfection and

efforts by the worm to avoid being killed. It is not clear whether the author was aware that these moves would cripple the attacked machines.



A "Buffer Overrun" Attack

In recent years, the novelty of vandalizing other people's computers has worn off some-what, and there are fewer jerks with programming skills who write new viruses. Other attacks by individuals with more criminal energy, whose intent has been to steal information or money, have surfaced. See [3] for a very readable account of the discovery and apprehension of one such person.

7.8 Regression Testing

It is a common and useful practice to make a new test whenever you find a program bug. You can use that test to verify that your bug fix really works. Don't throw the test away; feed it to the next version after that and all subsequent versions. Such a collection of test cases is called a *test suite*.

A test suite is a set of tests for repeated testing.

You will be surprised how often a bug that you fixed will reappear in a future version. This is a phenomenon known as *cycling*. Sometimes you don't quite understand the reason for a bug and apply a quick fix that appears to work. Later, you apply a different quick fix that solves a second problem but makes the first problem appear again. Of course, it is always best to think through what really causes a bug and fix the root cause instead of doing a sequence of “Band-Aid” solutions. If you don't succeed in doing that, however, you at least want to have an honest appraisal of how well the program works. By keeping all old test cases around and testing them against every new version, you get that feedback. The process of testing against a set of past failures is called *regression testing*.

Regression testing involves repeating previously run tests to ensure that known failures of prior versions do not appear in new versions of the software.

How do you organize a suite of tests? An easy technique is to produce multiple tester classes, such as `BankTester1, BankTester2`, and so on.

Another useful approach is to provide a generic tester, and feed it inputs from multiple files. Consider this tester for the `Bank` class of Section 7.5:

ch07/regression/BankTester.java

```
/**
   This program tests the Bank class.
*/
public class BankTester
{
   public static void main(String[] args)
   {
      Bank firstBankOfJava = new Bank();
      firstBankOfJava.addAccount(new BankAccount(1001, 20000));
      firstBankOfJava.addAccount(new BankAccount(1015, 10000));
      firstBankOfJava.addAccount(new BankAccount(1729, 15000));

      Scanner in = new Scanner(System.in);

      double threshold = in.nextDouble();
      int c = firstBankOfJava.count(threshold);
      System.out.println("Count: " + c);
      int expectedCount = in.nextInt();
      System.out.println("Expected: " + expectedCount);

      int accountNumber = in.nextInt;
      BankAccount a = firstBankOfJava.find(accountNumber);
      if (a == null)
         System.out.println("No matching account");
      else
      {
         System.out.println("Balance of maatching account: "
               + a.getBalance());
         int matchingBalance = in.nextLine();
         System.out.println("Expected: " + matchingBalance);
      }
   }
}
```

Rather than using fixed values for the threshold and the account number to be found, the program reads these values, and the expected responses. By running the program with different inputs, we can test different scenarios, such as the ones for diagnosing off-by-one errors discussed in Common Error 6.2.

Of course, it would be tedious to type in the input values by hand every time the test is executed. It is much better to save the inputs in a file, such as the following:

ch07/regression/input1.txt

```
15000
2
1015
10000
```

The command line interfaces of most operating systems provide a way to link a file to the input of a program, as if all the characters in the file had actually been typed by a user. Type the following command into a shell window:

```
java BankTester < input1.txt
```

The program is executed, but it no longer reads input from the keyboard. Instead, the `System.in` object (and the `Scanner` that reads from `System.in`) gets the input from the file `input1.txt`. This process is called *input redirection*.

The output is still displayed in the console window:

Output

```
Count: 2
Expected: 2
Balance of matching account: 10000
Expected: 10000
```

You can also redirect output. To capture the output of a program in a file, use the command

```
java BankTester < input1.txt > output1.txt
```

This is useful for archiving test cases.

SELF CHECK

15. Suppose you modified the code for a method. Why do you want to repeat tests that already passed with the previous version of the code?

16. Suppose a customer of your program finds an error. What action should you take beyond fixing the error?

17. Why doesn't the `BankTester` program contain prompts for the inputs?

PRODUCTIVITY HINT 7.1: Batch Files and Shell Scripts

If you need to perform the same tasks repeatedly on the command line, then it is worth learning about the automation features offered by your operating system.

Under Windows, you use batch files to execute a number of commands automatically. For example, suppose you need to test a program by running three testers:

```
java BankTester1
java BankTester2
java BankTester3 < input1.txt
```

Then you find a bug, fix it, and run the tests again. Now you need to type the three commands once more. There has to be a better way. Under Windows, put the commands in a text file and call it `test.bat`:

File test.bat

```
java BankTester1
java BankTester2
java BankTester3 < input1.txt
```

Then you just type

```
test.bat
```

and the three commands in the batch file execute automatically.

Batch files are a feature of the operating system, not of Java. On Linux, Mac OS, and UNIX, shell scripts are used for the same purpose. In this simple example, you can execute the commands by typing

```
sh test.bat
```

There are many uses for batch files and shell scripts, and it is well worth it to learn more about advanced features such as parameters and loops.

321

RANDOM FACT 7.2: The Therac-25 Incidents

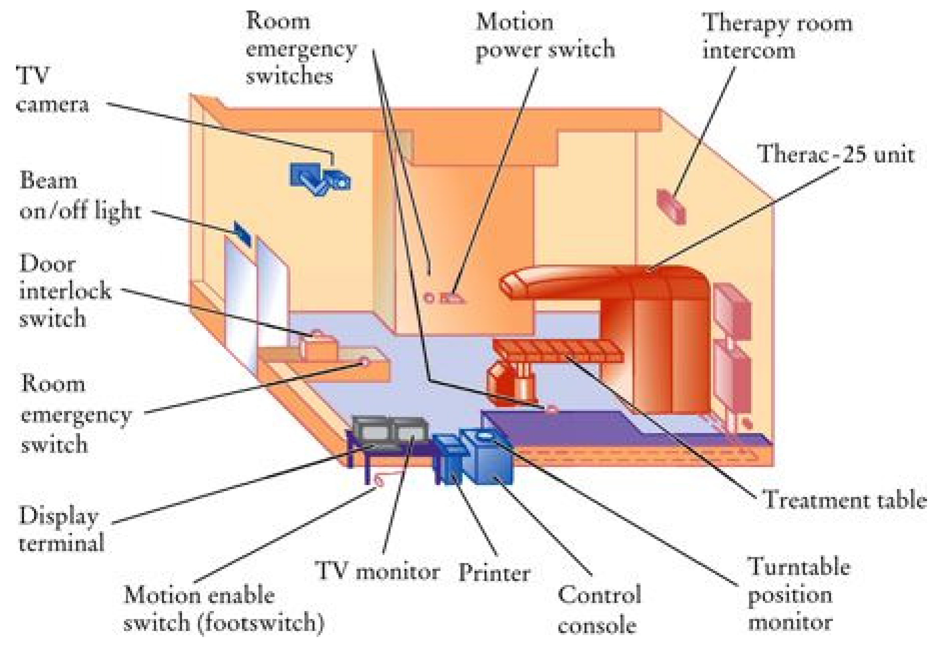
The Therac-25 is a computerized device to deliver radiation treatment to cancer patients (see Typical Therac-25 Facility). Between June 1985 and January 1987, several of these machines delivered serious overdoses to at least six patients, killing some of them and seriously maiming the others.

The machines were controlled by a computer program. Bugs in the program were directly responsible for the overdoses. According to Leveson and Turner [4], the program was written by a single programmer, who had since left the manufacturing company producing the device and could not be located. None of the company employees interviewed could say anything about the educational level or qualifications of the programmer.

The investigation by the federal Food and Drug Administration (FDA) found that the program was poorly documented and that there was neither a specification document nor a formal test plan. (This should make you think. Do you have a formal test plan for your programs?)

The overdoses were caused by an amateurish design of the software that had to control different devices concurrently, namely the keyboard, the display, the printer, and of course the radiation device itself. Synchronization and data sharing between the tasks were done in an ad hoc way, even though safe multitasking techniques were known at the time. Had the programmer enjoyed a formal education that involved these techniques, or taken the effort to study the literature, a safer machine could have been built. Such a machine would have probably involved a commercial multitasking system, which might have required a more expensive computer.

The same flaws were present in the software controlling the predecessor model, the Therac-20, but that machine had hardware interlocks that mechanically prevented overdoses.



Typical Therac-25 Facility

The hardware safety devices were removed in the Therac-25 and replaced by checks in the software, presumably to save cost.

Frank Houston of the FDA wrote in 1985: “A significant amount of software for life-critical systems comes from small firms, especially in the medical device industry; firms that fit the profile of those resistant to or uninformed of the principles of either system safety or software engineering” [4].

Who is to blame? The programmer? The manager who not only failed to ensure that the programmer was up to the task but also didn't insist on comprehensive testing? The hospitals that installed the device, or the FDA, for not reviewing the design process? Unfortunately, even today there are no firm standards of what constitutes a safe software design process.

CHAPTER SUMMARY

1. An array is a sequence of values of the same type.

2. You access array elements with an integer index, using the notation `a[i]`.
3. Index values of an array range from `0` to `length - 1`. Accessing a nonexistent element results in a bounds error.
4. Use the `length` field to find the number of elements in an array.
5. The `ArrayList` class manages a sequence of objects.
6. The `ArrayList` class is a generic class: `ArrayList<T>` collects objects of type `T`.
7. To treat primitive type values as objects, you must use wrapper classes.
8. The enhanced `for` loop traverses all elements of a collection.
9. To count values in an array list, check all elements and count the matches until you reach the end of the array list.
10. To find a value in an array list, check all elements until you have found a match.
11. To compute the maximum or minimum value of an array list, initialize a candidate with the starting element. Then compare the candidate with the remaining elements and update it if you find a larger or smaller value.
12. Two-dimensional arrays form a tabular, two-dimensional arrangement. You access elements with an index pair `a[i][j]`.
13. An array variable stores a reference to the array. Copying the variable yields a second reference to the same array.
14. Use the `clone` method to copy the elements of an array.

15. Use the `System.arraycopy` method to copy elements from one array to another.
16. Avoid parallel arrays by changing them into arrays of objects.
17. A test suite is a set of tests for repeated testing.

18. Regression testing involves repeating previously run tests to ensure that known failures of prior versions do not appear in new versions of the software.

FURTHER READING

1. Barton P. Miller, Louis Fericksen, and Bryan So, "An Empirical Study of the Reliability of Unix Utilities", *Communications of the ACM*, vol. 33, no. 12 (December 1990), pp. 32–44.

2. Peter J. Denning, *Computers under Attack*, Addison-Wesley, 1990.

3. Cliff Stoll, *The Cuckoo's Egg*, Doubleday, 1989.

4. Nancy G. Leveson and Clark S. Turner, "An Investigation of the Therac-25 Accidents," *IEEE Computer*, July 1993, pp. 18–41.

CLASSES, OBJECTS, AND METHODS INTRODUCED IN THIS CHAPTER

```
java.lang.Boolean
   booleanValue
java.lang.Double
   doubleValue
java.lang.Integer
   intValue
java.lang.System
   arraycopy
java.util.ArrayList<E>
   add
   get
   remove
   set
   size
```

REVIEW EXERCISES

★ **Exercise R7.1.** What is an index? What are the bounds of an array or array list? What is a bounds error?

★ **Exercise R7.2.** Write a program that contains a bounds error. Run the program. What happens on your computer? How does the error message help you locate the error?

★★ **Exercise R7.3.** Write Java code for a loop that simultaneously computes the maximum and minimum values of an array list. Use an array list of accounts as an example.

★ **Exercise R7.4.** Write a loop that reads 10 strings and inserts them into an array list. Write a second loop that prints out the strings in the opposite order from which they were entered.

★★ **Exercise R7.5.** Consider the algorithm that we used for determining the maximum value in an array list. We set `largestYet` to the starting element, which meant that we were no longer able to use the "for each" loop. An alternate approach is to initialize `largestYet` with `null`, then loop through all elements. Of course, inside the loop you need to test whether `largestYet` is still `null`. Modify the loop that finds the bank account with the largest balance, using this technique. Is this approach more or less efficient than the one used in the text?

★★★ **Exercise R7.6.** Consider another variation of the algorithm for determining the maximum value. Here, we compute the maximum value of an array of numbers.

```
double max = 0;// Contains an error!
for (x : values)
{
   if (x > max) max = x;
}
```

However, this approach contains a subtle error. What is the error, and how can you fix it?

★ **Exercise R7.7.** For each of the following sets of values, write code that fills an array a with the values.

a. `1 2 3 4 5 6 7 8 9 10`

b. `0 2 4 6 8 10 12 14 16 18 20`

c. `1 4 9 16 25 36 49 64 81 100`

d. `0 0 0 0 0 0 0 0 0 0`

e. `1 4 9 16 9 7 4 9 11`

Use a loop when appropriate.

★★ **Exercise R7.8.** Write a loop that fills an array a with 10 random numbers between 1 and 100. Write code (using one or more loops) to fill a with 10 different random numbers between 1 and 100.

★ **Exercise R7.9.** What is wrong with the following loop?

```
double[] data = new double[10];
for (int i = 1; i <= 10; i++) data[i] = i * i;
```

Explain two ways of fixing the error.

★★★ **Exercise R7.10.** Write a program that constructs an array of 20 integers and fills the first ten elements with the numbers 1, 4, 9, …, 100. Compile it and launch the debugger. After the array has been filled with three numbers, inspect it. What are the contents of the elements in the array beyond those that you filled?

★★ **Exercise R7.11.** Rewrite the following loops without using the "for each" construct. Here, `data` is an array of `double` values.

a. `for (x : data) sum = sum + x;`

b. `for (x : data) if (x == target) return true;`

c. `int i = 0;`

```
for (x : data) { data [i] = 2 * x; i++; }
```

★★ **Exercise R7.12.** Rewrite the following loops, using the "for each" construct. Here, `data` is an array of `double` values.

a. `for (int i = 0; i < data.length; i++) sum = sum + data[i];`

b. `for (int i = 1; i < data.length; i++) sum = sum + data[i];`

c.
```
for (int i = 0; i < data.length; i++)
    if (data[i] == target) return i;
```

★★★ **Exercise R7.13.** Give an example of

a. A useful method that has an array of integers as a parameter that is not modified.

b. A useful method that has an array of integers as a parameter that is modified.

c. A useful method that has an array of integers as a return value.

Describe each method; don't implement the methods.

★★★ **Exercise R7.14.** A method that has an array list as a parameter can change the contents in two ways. It can change the contents of individual array elements, or it can rearrange the elements. Describe two useful methods with `ArrayList<BankAccount>` parameters that change an array list of `BankAccount` objects in each of the two ways just described.

★ **Exercise R7.15.** What are parallel arrays? Why are parallel arrays indications of poor programming? How can they be avoided?

★★ **Exercise R7.16.** How do you perform the following tasks with arrays in Java?

a. Test that two arrays contain the same elements in the same order

b. Copy one array to another

c. Fill an array with zeroes, overwriting all elements in it

d. Remove all elements from an array list

★ **Exercise R7.17.** True or false?

- **a.** All elements of an array are of the same type.
- **b.** Array subscripts must be integers.
- **c.** Arrays cannot contain string references as elements.
- **d.** Arrays cannot use strings as subscripts.
- **e.** Parallel arrays must have equal length.
- **f.** Two-dimensional arrays always have the same numbers of rows and columns.

- **g.** Two parallel arrays can be replaced by a two-dimensional array.
- **h.** Elements of different columns in a two-dimensional array can have different types.

★★ **Exercise R7.18.** True or false?

- **a.** A method cannot return a two-dimensional array.
- **b.** A method can change the length of an array parameter.
- **c.** A method can change the length of an array list that is passed as a parameter.
- **d.** An array list can hold values of any type.

★T **Exercise R7.19.** Define the terms *regression testing* and *test suite*.

★T **Exercise R7.20.** What is the debugging phenomenon known as *cycling?* What can you do to avoid it?

Additional review exercises are available in WileyPLUS.

PROGRAMMING EXERCISES

★ **Exercise P7.1.** Add the following methods to the `Bank` class:

```
public void addAccount(int accountNumber,
double initialBalance)
public void deposit(int accountNumber, double
amount)
public void withdraw(int accountNumber,
double amount)
public double getBalance(int accountNumber)
```

★ **Exercise P7.2.** Implement a class `Purse`. A purse contains a collection of coins. For simplicity, we will only store the coin names in an `ArrayList<String>`. (We will discuss a better representation in Chapter 8.) Supply a method

```
void addCoin(String coinName)
```

Add a method `toString` to the `Purse` class that prints the coins in the purse in the format

```
Purse[Quarter,Dime,Nickel,Dime]
```

★ **Exercise P7.3.** Write a method reverse that reverses the sequence of coins in a purse. Use the toString method of the preceding assignment to test your code. For example, if reverse is called with a purse

```
Purse[Quarter,Dime,Nickel,Dime]
```

then the purse is changed to

```
Purse[Dime,Nickel,Dime,Quarter]
```

★ **Exercise P7.4.** Add a method to the Purse class

```
public void transfer(Purse other)
```

that transfers the contents of one purse to another. For example, if `a` is

```
Purse[Quarter,Dime,Nickel,Dime]
```

and `b` is

```
Purse[Dime,Nickel]
```

then after the call `a.transfer(b), a` is

```
Purse[Quarter,Dime,Nickel,Dime,Dime,Nickel]
```

and `b` is empty.

★ **Exercise P7.5.** Write a method for the `Purse` class

```
public boolean sameContents(Purse other)
```

that checks whether the other purse has the same coins in the same order.

★★ **Exercise P7.6.** Write a method for the `Purse` class

```
public boolean sameCoins(Purse other)
```

that checks whether the other purse has the same coins, perhaps in a different order. For example, the purses

```
Purse[Quarter,Dime,Nickel,Dime]
```

and

```
Purse[Nickel,Dime,Dime,Quarter]
```

should be considered equal.

You will probably need one or more helper methods.

★★ **Exercise P7.7.** A `Polygon` is a closed curve made up from line segments that join the polygon's corner points. Implement a class `Polygon` with methods

```
public double perimeter()
```

and

```
public double area()
```

that compute the circumference and area of a polygon. To compute the perimeter, compute the distance between adjacent points, and total up the distances. The area of a polygon with corners $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ is

$$\frac{1}{2}(x_0 y_0 + x_1 y_2 + \cdots + x_{n-1} y_0 - y_0 x_1 - y_1 x_2 - \cdots - y_{n-1} x_0)$$

As test cases, compute the perimeter and area of a rectangle and of a regular hexagon. *Note*: You need not draw the polygon—that is done in Exercise P7.15.

★ **Exercise P7.8.** Write a program that reads a sequence of integers into an array and that computes the alternating sum of all elements in the array. For example, if the program is executed with the input data

1 4 9 16 9 7 4 9 11

then it computes

$$1 - 4 + 9 - 16 + 9 - 7 + 4 - 9 + 11 = -2$$

PROGRAMMING EXERCISES

★★ **Exercise P7.9.** Write a program that produces random permutations of the numbers 1 to 10. To generate a random permutation, you need to fill an array with the numbers 1 to 10 so that no two entries of the array have the same contents. You could do it by brute force, by calling `Random.nextInt` until it produces a value that is not yet in the array. Instead, you should implement a smart method. Make a second array and fill it with the numbers 1 to 10. Then pick one of those at random, remove it, and append it to the permutation array. Repeat 10 times. Implement a class `Permutation Generator` with a method

```
int[] nextPermutation
```

★★ **Exercise P7.10.** Add a method `getWinner` to the `TicTacToe` class of Section 7.6. It should return `"x"` or `"o"` to indicate a winner, or `"  "` if there is no winner yet. Recall that a winning position has three matching marks in a row, column, or diagonal.

★★★ **Exercise P7.11.** Write an application that plays tic-tac-toe. Your program should draw the game board, change players after every successful move, and pronounce the winner.

★★ **Exercise P7.12.** *Magic squares*. An $n \times n$ matrix that is filled with the numbers 1, 2, 3, …, n^2 is a magic square if the sum of the elements in each row, in each column, and in the two diagonals is the same value. For example,

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

Write a program that reads in n^2 values from the keyboard and tests whether they form a magic square when arranged as a square matrix. You need to test three features:

- Did the user enter n^2 numbers for some n?
- Do each of the numbers 1, 2, …, n^2 occur exactly once in the user input?
- When the numbers are put into a square, are the sums of the rows, columns, and diagonals equal to each other?

If the size of the input is a square, test whether all numbers between 1 and n^2 are present. Then compute the row, column, and diagonal sums. Implement a class `Square` with methods

```
public void add(int i)
public boolean isMagic()
```

★★ **Exercise P7.13.** Implement the following algorithm to construct magic n-by-n^2 squares; it works only if n is odd. Place a 1 in the middle of the bottom row. After k has been placed in the (i, j) square, place $k + 1$ into the square to the right and down, wrapping around the borders. However, if the square to the right and down has already been filled, or if you are in the lower-right corner, then you must move to the square straight up instead. Here is the 5 × 5 square that you get if you follow this method: 329 330

11	18	25	2	9
10	12	19	21	3
4	6	13	20	22
23	5	7	14	16
17	24	1	8	15

Write a program whose input is the number n and whose output is the magic square of order n if n is odd. Implement a class `MagicSquare` with a constructor that constructs the square and a `toString` method that returns a representation of the square.

★G **Exercise P7.14.** Implement a class Cloud that contains an array list of Point `2D.Double` objects. Support methods

```
public void add(Point2D.Double aPoint)
public void draw(Graphics2D g2)
```

Draw each point as a tiny circle.

Write a graphical application that draws a cloud of 100 random points.

★★G **Exercise P7.15.** Implement a class `Polygon` that contains an array list of `Point2D.Double objects.` Support methods

```
public void add(Point2D.Double aPoint)
public void draw(Graphics2D g2)
```

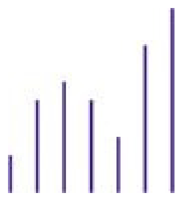
Draw the polygon by joining adjacent points with a line, and then closing it up by joining the end and start points.

Write a graphical application that draws a square and a pentagon using two `Polygon` objects.

★G **Exercise P7.16.** Write a class Chart with methods

```
public void add(int value)
public void draw(Graphics2D g2)
```

that displays a stick chart of the added values, like this:



You may assume that the values are pixel positions.

★★G **Exercise P7.17.** Write a class BarChart with methods

```
public void add(double value)
public void draw(Graphics2D g2)
```

that displays a chart of the added values. You may assume that all values in data are positive. Stretch the bars so that they fill the entire area of the screen. You must figure out the maximum of the values, and then scale each bar.

★★★G **Exercise P7.18.** Improve the `BarChart` class of Exercise P7.17 to work correctly when the data contains negative values.

★★G **Exercise P7.19.** Write a class `PieChart` with methods

```
public void add (double value)
public void draw(Graphics2D g2)
```

that displays a pie chart of the values in `data`. You may assume that all data values are positive.

Additional programming exercises are available in WileyPLUS.

PROGRAMMING PROJECTS

★★★ **Project 7.1.** *Poker Simulator*. In this assignment, you will implement a simulation of a popular casino game usually called video poker. The card deck contains 52 cards, 13 of each suit. At the beginning of the game, the deck is shuffled. You need to devise a fair method for shuffling. (It does not have to be efficient.) Then the top five cards of the deck are presented to the player. The player can reject none, some, or all of the cards. The rejected cards are replaced from the top of the deck. Now the hand is scored. Your program should pronounce it to be one of the following:

- No pair—The lowest hand, containing five separate cards that do not match up to create any of the hands below.
- One pair—Two cards of the same value, for example two queens.
- Two pairs—Two pairs, for example two queens and two 5's.
- Three of a kind—Three cards of the same value, for example three queens.
- Straight—Five cards with consecutive values, not necessarily of the same suit, such as 4, 5, 6, 7, and 8. The ace can either precede a 2 or follow a king.
- Flush—Five cards, not necessarily in order, of the same suit.
- Full House—Three of a kind and a pair, for example three queens and two 5's
- Four of a Kind—Four cards of the same value, such as four queens.
- Straight Flush—A straight and a flush: Five cards with consecutive values of the same suit.

- Royal Flush—The best possible hand in poker. A 10, jack, queen, king, and ace, all of the same suit.

If you are so inclined, you can implement a wager. The player pays a JavaDollar for each game, and wins according to the following payout chart:

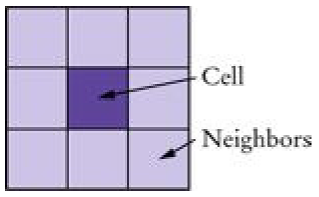
Hand	Payout	Hand	Payout
Royal Flush	250	Straight	4
Straight Flush	50	Three of a Kind	3
Four of a Kind	25	Two Pair	2
Full House	6	Pair of Jacks or Better	1
Flush	5		

★★★ **Project 7.2.** *The Game of Life* is a well-known mathematical game that gives rise to amazingly complex behavior, although it can be specified by a few simple rules. (It is not actually a game in the traditional sense, with players competing for a win.) Here are the rules. The game is played on a rectangular board. Each square can be either empty or occupied. At the beginning, you can specify empty and occupied cells in some way; then the game runs automatically. In each *generation*, the next generation is computed. A new cell is born on an empty square if it is surrounded by exactly three occupied neighbor cells. A cell dies of overcrowding if it is surrounded by four or more neighbors, and it dies of loneliness if it is surrounded by zero or one neighbor. A neighbor is an occupant of an adjacent square to the left, right, top, or bottom or in a diagonal direction. Figure 13 shows a cell and its neighbor cells.

Many configurations show interesting behavior when subjected to these rules. Figure 14 shows a *glider*, observed over five generations. Note how it moves. After four generations, it is transformed into the identical shape, but located one square to the right and below.

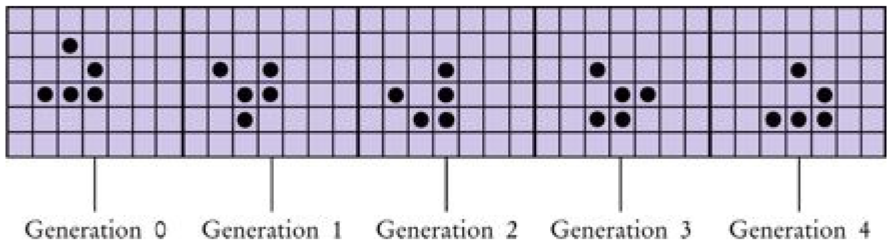
One of the more amazing configurations is the glider gun: a complex collection of cells that, after 30 moves, turns back into itself and a glider (see Figure 15).

Figure 13



Neighborhood of a Cell in the Game of Life

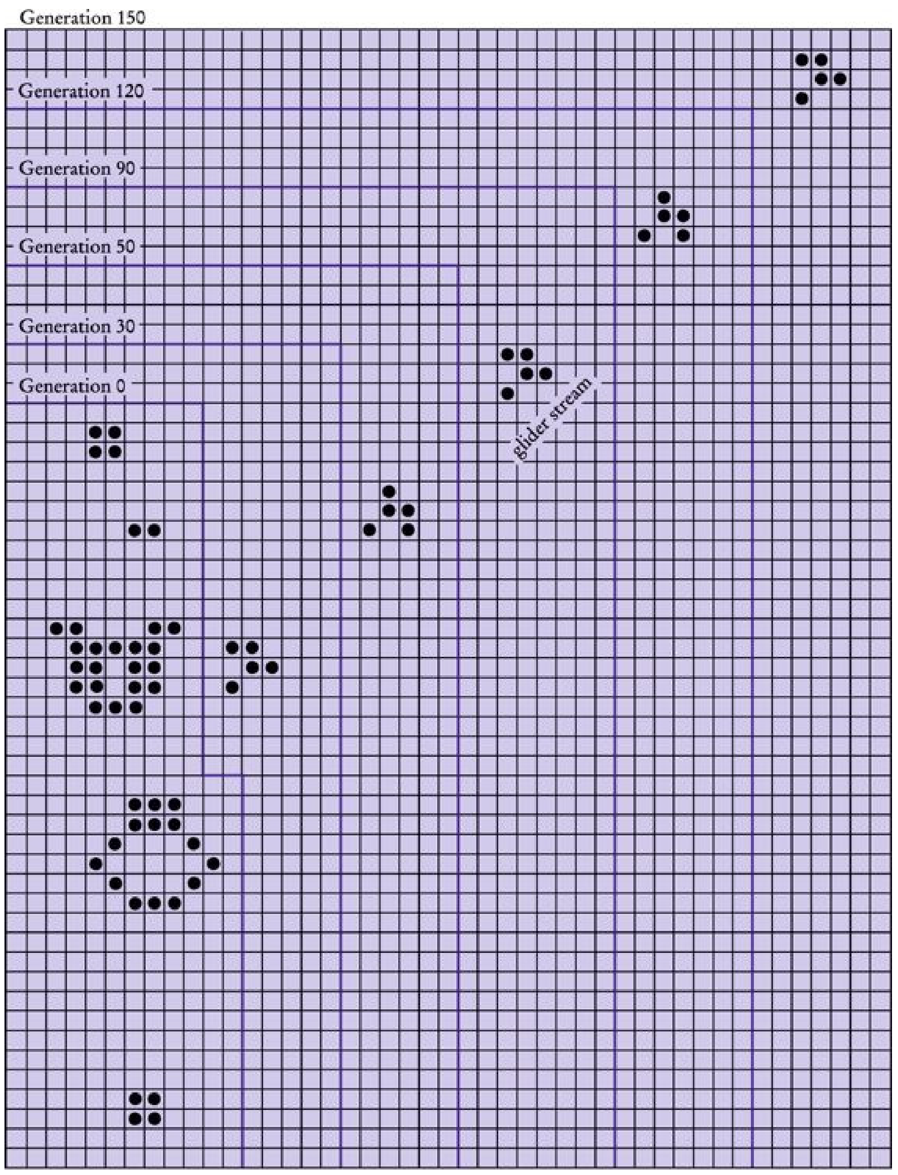
Figure 14



Glider

332

Figure 15



Glider Gun

Program the game to eliminate the drudgery of computing successive generations by hand. Use a two-dimensional array to store the rectangular configuration. Write a program that shows successive generations of the game. You may get extra credit if you implement a graphical application that allows the user to add or remove cells by clicking with the mouse.

ANSWERS TO SELF-CHECK QUESTIONS

1. 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, but *not* 100.

2.

- **a.** 0
- **b.** a run-time error: array index out of bounds
- **c.** a compile-time error: `c` is not initialized

3. `new String[10];`

`new ArrayList<String>();`

4. `names` contains the strings `"B"` and `"C"` at positions 0 and 1.

5. `double` is one of the eight primitive types. `Double` is a class type.

6. `data.set(0, data.get(0) + 1);`

7. `for (double x : data) System.out.println(x);`

8. The loop writes a value into `data[i]`. The “for each” loop does not have the index variable `i`.

9. It returns the first match that it finds.

10. Yes, but the first comparison would always fail.

11. `int[][] array = new int[4][4];`

12. `int count = 0;`

```
for (int i = 0; i < ROWS; i++)
   for (int j = 0; j < COLUMNS; j++)
      if (board[i][j].equals(" ")) count++;
```

13. Use the `add` and `remove` methods.

14. Allocating a new array and copying the elements is time-consuming. You wouldn't want to go through the process every time you add an element.

15. It is possible to introduce errors when modifying code.

16. Add a test case to the test suite that verifies that the error is fixed.

17. There is no human user who would see the prompts because input is provided from a file.