

CHAPTER 9

Advanced Modularization Techniques

After studying this chapter, you will be able to:

- ◎ Write methods that require no parameters
- ◎ Write methods that require a single parameter
- ◎ Write methods that require multiple parameters
- ◎ Write methods that return values
- ◎ Pass entire arrays and single elements of an array to a method
- ◎ Overload methods
- ◎ Use Java's built-in methods

In Chapter 2 of *Programming Logic and Design, Eighth Edition*, you learned that local variables are variables that are declared within the method that uses them. You also learned that most programs consist of a main method, which contains the mainline logic and calls other methods to get specific work done in the program.

In this chapter, you learn more about methods in Java. You learn how to write methods that require no parameters, how to write methods that require a single parameter, how to write methods that require multiple parameters, and how to write methods that return a value. You also learn how to pass an array to a method, how to overload a method, and how to use some of Java's built-in methods. To help you learn about methods, you will study some Java programs that implement the logic and design presented in *Programming Logic and Design*.

You should do the exercises and labs in this chapter after you have finished Chapter 9 of *Programming Logic and Design*.

Writing Methods with No Parameters

To begin learning about methods, we review the Java code for a Customer Bill program, shown in Figure 9-1. Notice the line numbers in front of each line of code in this program. These line numbers are not actually part of the program but are included for reference only.

```
1 import javax.swing.JOptionPane;
2 public class CustomerBill
3 {
4     public static void main(String args[])
5     {
6         // Declare variables local to main()
7         String name;
8         String balanceString;
9         double balance;
10
11         // Get interactive input
12         name = JOptionPane.showInputDialog(
13             "Enter customer's name: ");
14         balanceString = JOptionPane.showInputDialog(
15             "Enter customer's balance: ");
16
17         // Convert String to double
18         balance = Double.parseDouble(balanceString);
19
20         // Call nameAndAddress() method
```

Figure 9-1 Java code for the Customer Bill program (*continues*)

(continued)

```
21     nameAndAddress();
22
23     // Output customer name and address
24     System.out.println("Customer Name: " + name);
25     System.out.println("Customer Balance: " + balance);
26
27 }
28 public static void nameAndAddress()
29 {
30     // Declare and initialize local, constant Strings
31     final String ADDRESS_LINE1 = "ABC Manufacturing";
32     final String ADDRESS_LINE2 = "47 Industrial Lane";
33     final String ADDRESS_LINE3 = "Wild Rose, WI 54984";
34
35     // Output
36     System.out.println(ADDRESS_LINE1);
37     System.out.println(ADDRESS_LINE2);
38     System.out.println(ADDRESS_LINE3);
39 } // End of nameAndAddress() method
40 }
```

Figure 9-1 Java code for the Customer Bill program

The program begins execution with the `main()` method, which is shown on line 4. This method contains the declaration of three variables (lines 7, 8, and 9), `name`, `balanceString`, and `balance`, which are local to the `main()` method. Next, on lines 12, 13, 14, and 15, interactive input statements retrieve values for `name` and `balanceString`, and on line 18 `balanceString` is converted to the `double` data type. The method `nameAndAddress()` is then called on line 21, with no arguments listed within its parentheses. **Arguments**, which are sometimes called **actual parameters**, are data items sent to methods. There are no arguments for the `nameAndAddress()` method because this method requires no data. You will learn about passing arguments to methods later in this chapter. The last two statements (lines 24 and 25) in the `main()` method are print statements that output the customer's name and balance.

Next, on line 28, you see the header for the `nameAndAddress()` method. The **header** begins with the `public` keyword, followed by the `static` keyword, followed by the `void` keyword, followed by the method name, which is `nameAndAddress()`. The `public` keyword makes this method available for execution. The keyword `static` means you do not have to create a `CustomerBill` object to call the method, and the `void` keyword indicates that the `nameAndAddress()` method does not return a value. You learn more about methods that return values later in this chapter. In the next part of the Customer Bill program, you see three constants that are local to the `nameAndAddress()` method: `ADDRESS_LINE1`,

ADDRESS_LINE2, and ADDRESS_LINE3. These constants are declared and initialized on lines 31, 32, and 33 and then printed on lines 36, 37, and 38. When the input to this program is Ed Gonzales (name) and 352.39 (balance), the output is shown in Figure 9-2.



```
C:\Java>java CustomerBill
ABC Manufacturing
47 Industrial Lane
Wild Rose, WI 54984
Customer Name: Ed Gonzales
Customer Balance: 352.39
C:\Java>
```

Figure 9-2 Output from the Customer Bill program

Exercise 9-1: Writing Methods with No Parameters

In this exercise, you use what you have learned about writing methods with no parameters to answer Questions 1–2.

1. Given the following method calls, write the method's header:

a. `printMailingLabel();`

b. `displayOrderNumbers();`

c. `displayTVListing();`

2. Given the following method headers, write a method call:

a. `public static void printCellPhoneNumbers()`

b. `public static void displayTeamMembers()`

c. `public static void showOrderInfo()`

Lab 9-1: Writing Methods with No Parameters

In this lab, you complete a partially prewritten Java program that includes a method with no parameters. The program asks the user if they have preregistered for art show tickets. If the user has preregistered, the program should call a method named `discount()` that displays the message "You are preregistered and qualify for a 5% discount." If the user has not preregistered, the program should call a method named `noDiscount()` that displays the message "Sorry, you did not preregister and do not qualify for a 5% discount." The source code file provided for this lab includes the necessary variable declarations and the input statement. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `ArtShow.java` using Notepad or the text editor of your choice.
2. Write the Java statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file, `ArtShow.java`.
5. Execute the program.

Writing Methods that Require a Single Parameter

As you learned in *Programming Logic and Design*, some methods require data to accomplish their task. You also learned that designing a program so it sends data (which can be different each time the program runs) to a method (which does not change) keeps you from having to write multiple methods to handle similar situations. For example, suppose you are writing a program that has to determine if a number is even or odd. It is certainly better to write a single method, to which the program can pass a number entered by the user, than to write individual methods for every number.

In Figure 9-3, you see the Java code for a program that includes a method that can determine if a number is odd or even. The line numbers are not actually part of the program but are included for reference only. The program allows the user to enter a number, and then passes that number to a method as an argument. After it receives the argument, the method can determine if the number is an even number or an odd number.

```

1 // EvenOrOdd.java - This program determines if a number
2 // input by the user is an even number or an odd number.
3 // Input: Interactive.
4 // Output: The number entered and whether it is even or odd.
5
6 import javax.swing.*;
7
8 public class EvenOrOdd
9 {
10     public static void main(String args[])
11     {
12         int number;
13         String numberString;
14
15         numberString = JOptionPane.showInputDialog(
16             "Enter a number or -999 to quit: ");
17         number = Integer.parseInt(numberString);
18
19         while(number != -999)
20         {
21             even_or_odd(number);
22             numberString = JOptionPane.showInputDialog(
23                 "Enter a number or -999 to quit: ");
24             number = Integer.parseInt(numberString);
25         }
26
27         System.exit(0);
28     } // End of main() method.
29
30     public static void even_or_odd(int number)
31     {
32         if((number % 2) == 0)
33             System.out.println("Number: " + number +
34                               " is even.");
35         else
36             System.out.println("Number: " + number +
37                               " is odd.");
38     } // End of even_or_odd method.
39 } // End of EvenOrOdd class.

```

The variable named `number` is local to the `main()` method. Its value is stored at one memory location. For example, it may be stored at memory location 2000.

The value of the formal parameter, `number`, is stored at a different memory location and is local to the `even_or_odd()` method. For example, it may be stored at memory location 7800.

Figure 9-3 Java code for the Even or Odd program

On lines 15 and 16 in this program, the user is asked to enter a number or the sentinel value, -999, when she is finished entering numbers and wants to quit the program. (You learned about sentinel values in Chapter 5 of this book.) On lines 15, 16, and 17, the input value is retrieved, stored in the variable named `numberString`, converted to an `int`, and then stored in the variable named `number`. Next, if the user did not enter the sentinel value -999, the `while` loop is entered, and the method named `even_or_odd()` is called (line 21) using the following syntax, `even_or_odd(number);`.

Notice that the variable, `number`, is placed within the parentheses on line 21, which means that the value of `number` is passed to the `even_or_odd()` method. This is referred to as passing an argument by value. **Passing an argument by value** means that a copy of the value of the argument is passed to the method. Within the method, the value is stored in the formal parameter at a different memory location, and is considered local to that method. In this example, as shown on line 31, the value is stored in the formal parameter named `number`.

Program control is now transferred to the `even_or_odd()` method. The header for the `even_or_odd()` method on line 31 includes the `public`, `static`, and `void` keywords, as discussed earlier in this chapter. The name of the method follows, and within the parentheses that follow the method name, the parameter, `number`, is given a local name and declared as the `int` data type.



The data type of the formal parameter and the actual parameter must be the same.

Remember that even though the name of the parameter, `number`, has the same name as the local variable `number` in the `main()` method, they are stored at different memory locations. Figure 9-3 shows that the variable `number` that is local to `main()` is stored at one memory location, and the parameter, `number`, in the `even_or_odd()` method is stored at a different memory location.

Within the method on line 33, the modulus operator (`%`) is used in the test portion of the `if` statement to determine if the value of the local `number` is even or odd. The user is then informed if `number` is even (lines 34 and 35) or odd (lines 37 and 38), and program control is transferred back to the statement that follows the call to `even_or_odd()` in the `main()` method (line 22).

Back in the `main()` method, the user is asked to enter another number on lines 22 and 23, and the `while` loop continues to execute, calling the `even_or_odd()` method with a new input value. The loop is exited when the user enters the sentinel value `-999`, and the program ends. When the input to this program is `45, 98, 1, -32, 643, 999` and `-999`, the output is shown in Figure 9-4.

```
C:\Java>java EvenOrOdd
Number: 45 is odd.
Number: 98 is even.
Number: 1 is odd.
Number: -32 is even.
Number: 643 is odd.
Number: 999 is odd.
C:\Java>
```

Figure 9-4 Output from the Even or Odd program

Exercise 9-2: Writing Methods that Require a Single Parameter

In this exercise, you use what you have learned about writing methods that require a single parameter to answer Questions 1–2.

1. Given the following variable declarations and method calls, write the method's header:

- a. `String name;`
`printNameBadge(name);`

- b. `double side_length;`
`calculateSquareArea(side_length);`

- c. `int hours;`
`displaySecondsInMinutes(minutes);`

2. Given the following method headers, write a method call:

- a. `String petName = "Mindre";`
`public static void displayPetName(String petName)`

- b. `int currentMonth;`
`public static void printBirthdays(int month)`

- c. `String password;`
`public static void checkValidPassword(String password)`

Lab 9-2: Writing Methods that Require a Single Parameter

In this lab, you complete a partially written Java program that includes two methods that require a single parameter. The program continuously prompts the user for an integer until the user enters 0. The program then passes the value to a method that computes the sum of all the whole numbers from 1 up to and including the entered number. Next, the program passes the value to another method that computes the product of all the whole numbers up to and including the entered number. The source code file provided for this lab includes the

necessary variable declarations and the input statement. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `SumAndProduct.java` using Notepad or the text editor of your choice.
2. Write the Java statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file, `SumAndProduct.java`.
5. Execute the program.

Writing Methods that Require Multiple Parameters

In Chapter 9 of *Programming Logic and Design*, you learned that a method often requires more than one parameter in order to accomplish its task. To specify that a method requires multiple parameters, you include a list of data types and local identifiers separated by commas as part of the method's header. To call a method that expects multiple parameters, you list the actual parameters (separated by commas) in the call to the method.

In Figure 9-5, you see the Java code for a program that includes a method named `computeTax()` that you designed in *Programming Logic and Design*. The line numbers are not actually part of the program but are included for reference only.

```
1 // ComputeTax.java - This program computes tax given a
2 // balance and a rate
3 // Input: Interactive.
4 // Output: The balance, tax rate, and computed tax.
5
6 import javax.swing.*;
7
8 public class ComputeTax
9 {
10     public static void main(String args[])
11     {
12         double balance; _____ Memory address 1000
13         String balanceString;
14         double rate; _____ Memory address 1008
15         String rateString;
16
17         balanceString = JOptionPane.showInputDialog(
18             "Enter balance: ");
19         balance = Double.parseDouble(balanceString);
20         rateString = JOptionPane.showInputDialog(
21             "Enter rate: ");
22         rate = Double.parseDouble(rateString);
23
24         computeTax(balance, rate);
25
26         System.exit(0);
27
28     } // End of main() method. _____ Memory address 9000
29
30     public static void computeTax(double amount, double rate)
31     { _____ Memory address 9008
32         double tax;
33
34         tax = amount * rate;
35         System.out.println("Amount: " + amount + " Rate: " +
36             rate + " Tax: " + tax);
37
38     } // End of computeTax method
39 } // End of ComputeTax class.
```

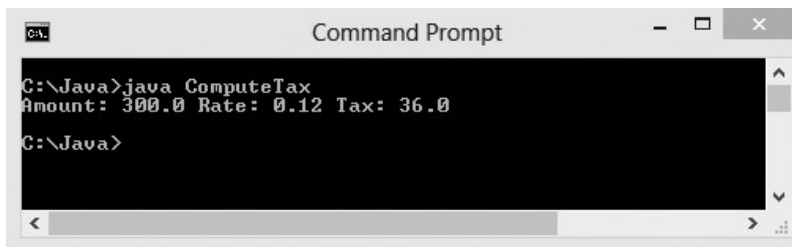
Figure 9-5 Java code for the Compute Tax program

In the Java code shown in Figure 9-5, you see that the highlighted call to `computeTax()` on line 24 includes the names of the local variables `balance` and `rate` within the parentheses and that they are separated by a comma. These are the arguments (actual parameters) that are passed to the `computeTax()` method. You can also see that the `computeTax()` method header on line 30 is highlighted and includes two formal parameters, `double amount` and `double rate`, listed within parentheses and separated by a comma. The value of the variable named `balance` is passed by value to the `computeTax()` method as an actual parameter and is stored

in the formal parameter named `amount`. The value of the variable named `rate` is passed by value to the `computeTax()` method as an actual parameter and is stored in the formal parameter named `rate`. As illustrated in Figure 9-5, it does not matter that one of the parameters being passed, `rate`, has the same name as the parameter received, `rate`, because they occupy different memory locations. When the input to this program is 300.00 (balance) and .12 (rate), the output is shown in Figure 9-6.



In Java, when you write a method that expects more than one argument, you must list the arguments separately, even if they have the same data type.



```
CA Command Prompt
C:\Java>java ComputeTax
Amount: 300.0 Rate: 0.12 Tax: 36.0
C:\Java>
```

Figure 9-6 Output from the Compute Tax program



There is no limit to the number of arguments you can pass to a method, but when multiple arguments are passed to a method, the call to the method and the method's header must match. This means that the number of arguments, their data types, and the order in which they are listed must be the same.

Exercise 9-3: Writing Methods that Require Multiple Parameters

In this exercise, you use what you have learned about writing methods that require multiple parameters to answer Questions 1–2.

1. Given the following method calls, write the method's header:

a. `String name, address;`
`printLabel(name, address);`

b. `double side1, side2;`
`calculateRectangleArea(side1, side2);`

c. `int day, month, year;`
`birthdayInvitation(day, month, year);`

2. Given the following method headers, write a method call:
 - a. `public static void printBill(String name, double balance)`
 - b. `public static void findProduct(int num1, int num2)`
 - c. `public static void newBalance(double bal, double percent)`

Lab 9-3: Writing Methods that Require Multiple Parameters

In this lab, you complete a partially written Java program that includes methods that require multiple parameters (arguments). The program prompts the user for two numeric values. Both values should be passed to methods named `calculateSum()`, `calculateDifference()`, and `calculateProduct()`. The methods compute the sum of the two values, the difference between the two values, and the product of the two values. Each method should perform the appropriate computation and display the results. The source code file provided for this lab includes the variable declarations and the input statements. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `Computation.java` using Notepad or the text editor of your choice.
2. Write the Java statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file, `Computation.java`.
5. Execute the program.

Writing Methods that Return a Value

Thus far in this book, none of the methods you have written return a value. The header for each of these methods includes the keyword `void`, as in, `public static void main()`, indicating that the method does not return a value. However, as a programmer, you will often find that you need to write methods that do return a value. In Java, a method can only return a single value; when you write the code for the method, you must indicate the data type of the value you want to return. This is often referred to as the method's return type. The return type can be any of Java's built-in data types, as well as a class type, such as `String`. You will learn more about classes in Chapter 10 of this book. For now, you will focus on returning values of the built-in types and `String` objects.

In Chapter 9 of *Programming Logic and Design*, you studied the design for a program that includes a method named `getHoursWorked()`. This method is designed to prompt a user for the number of hours an employee has worked, retrieve the value, and then return that value to the location in the program where the method was called. The Java code that implements this design is shown in Figure 9-7.

```
1 // GrossPay.java - This program computes an employee's
2 // gross pay.
3 // Input: Interactive.
4 // Output: The employee's hours worked and their gross pay.
5
6 import javax.swing.*;
7
8 public class GrossPay
9 {
10     public static void main(String args[])
11     {
12         double hours;
13         final double PAY_RATE = 12.00;
14         double gross;
15
16         hours = getHoursWorked();
17         gross = hours * PAY_RATE;
18
19         System.out.println("Hours worked: " + hours);
20         System.out.println("Gross pay is: " + gross);
21
22         System.exit(0);
23     }
24 }
```

Figure 9-7 Java code for a program that includes the `getHoursWorked()` method (*continues*)

(continued)

```
25
26 public static double getHoursWorked()
27 {
28     double workHours;
29     String workHoursString;
30
31     workHoursString = JOptionPane.showInputDialog(
32         "Please enter hours worked: ");
33     workHours = Double.parseDouble(workHoursString);
34
35     return workHours;
36 }
37 // End of getHoursWorked method
38 // End of GrossPay class.
```

Figure 9-7 Java code for a program that includes the `getHoursWorked()` method

The Java program shown in Figure 9-7 declares local constants and variables `hours`, `PAY_RATE`, and `gross` on lines 12, 13, and 14 in the `main()` method. The next statement (line 16), shown below, is an assignment statement.

```
hours = getHoursWorked();
```

This assignment statement includes a call to the method named `getHoursWorked()`. As with all assignment statements, the expression on the right side of the assignment operator (`=`) is evaluated, and then the result is assigned to the variable named on the left side of the assignment operator (`=`). In this example, the expression on the right is a call to the `getHoursWorked()` method.

When the `getHoursWorked()` method is called, program control is transferred to the method. Notice that the header (line 26) for this method is written as follows:

```
public static double getHoursWorked()
```

The keyword `double` is used in the header to specify that a value of data type `double` is returned by this method.

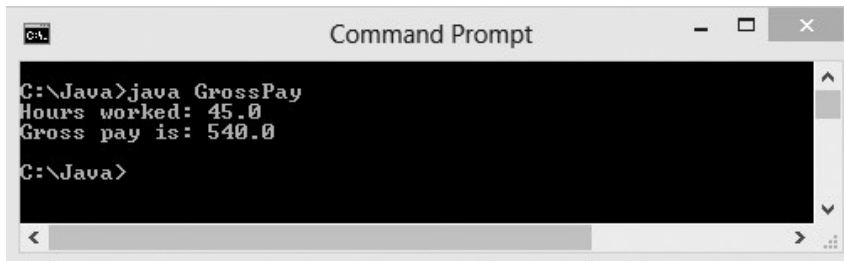
Two local variables, `workHours` (data type `double`) and `workHoursString` (a `String` object), are then declared on lines 28 and 29. On lines 31 and 32, the user is then asked to enter the number of hours worked, at which point the value is retrieved and stored in `workHoursString`. Next, on line 33, `workHoursString` is converted to a `double`, and assigned to the local variable named `workHours`. The return statement that follows on line 35 returns a copy of the value stored in `workHours` (data type `double`) to the location in the calling method where `getHoursWorked()` is called, which is the right side of the assignment statement on line 16.

The value returned to the right side of the assignment statement is then assigned to the variable named `hours` (data type `double`) in the `main()` method. Next, the gross pay is calculated on line 17, followed by the `System.out.println()` statements on lines 19 and 20 that display the value of the local variables, `hours` and `gross`, which contain the number of hours worked and the calculated gross pay.

You can also use a method's return value directly rather than store it in a variable. The two Java statements that follow make calls to the same `getHoursWorked()` method shown in Figure 9-7, but in these statements the returned value is used directly in the statement that calculates gross pay and in the statement that prints the returned value.

```
gross = getHoursWorked() * PAY_RATE;  
System.out.println("Hours worked is " + getHoursWorked());
```

When the input to this program is 45, the output is shown in Figure 9-8.



```
C:\Java>java GrossPay  
Hours worked: 45.0  
Gross pay is: 540.0  
C:\Java>
```

Figure 9-8 Output from program that includes the `getHoursWorked()` method

Exercise 9-4: Writing Methods that Return a Value

In this exercise, you use what you have learned about writing methods that return a value to answer Questions 1–2.

1. Given the following variable declarations and method calls, write the method's header:

a. `double price, percent, newPrice;`
`newPrice = calculateNewPrice(price, percent);`

b. `double area, one_length, two_length;`
`area = calcArea(one_length, two_length);`

```
c. String lowerCase, upperCase;  
   upperCase = changeCase(lowerCase);
```

2. Given the following method headers, write a method call:

```
a. public static String findItemType(int itemNumber)
```

```
b. public static int cubed(int num1, int num2)
```

```
c. public static int power(int num, int exp)
```

Lab 9-4: Writing Methods that Return a Value

In this lab, you complete a partially written Java program that includes a method that returns a value. The program is a simple calculator that prompts the user for two numbers and an operator (+, -, *, /, or %). The two numbers and the operator are passed to the method where the appropriate arithmetic operation is performed. The result is then returned to the `main()` method where the arithmetic operation and result are displayed. For example, if the user enters 3, 4, and *, the following is displayed:

```
3.00 * 4.00 = 12.00
```

The source code file provided for this lab includes the necessary variable declarations, and input and output statements. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `Calculator.java` using Notepad or the text editor of your choice.
2. Write the Java statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file, `Calculator.java`.
5. Execute the program.

Passing an Array and an Array Element to a Method

As a Java programmer, there are times when you will want to write a method that will perform a task on all of the elements you have stored in an array. For example, in Chapter 9 of *Programming Logic and Design*, you saw a design for a program that used a method to quadruple all of the values stored in an array. This design is translated into Java code in Figure 9-9.

The `main()` method begins on line 4 and proceeds with the declaration and initialization of the constant named `LENGTH` (line 7) and the array of integers named `someNums` (line 8), followed by the declaration of the variable named `x` (line 9), which is used as a loop control variable. The first `while` loop in the program on lines 12 through 16 prints the values stored in the array at the beginning of the program. On line 18, the method named `quadrupleTheValues()` is called. The array named `someNums` is passed as an argument.

```
1 import javax.swing.*;
2 public class PassEntireArray
3 {
4     public static void main(String args[])
5     {
6         // Declare variables
7         final int LENGTH = 4;
8         int someNums[] = {10, 12, 22, 35};
9         int x;
10        System.out.println("At beginning of main method...");
11        x = 0;
12        while (x < LENGTH) // Print initial array values
13        {
14            System.out.println(someNums[x]);
15            x++;
16        }
17        // Call method, pass array
18        quadrupleTheValues(someNums);
19        System.out.println("At the end of main method...");
20        x = 0;
21        // Print changed array values
22        while (x < someNums.length)
23        {
24            System.out.println(someNums[x]);
25            x++;
26        }
27        System.exit(0);
28    } // End of main() method.
```

Figure 9-9 Java code for the Pass Entire Array program (*continues*)

(continued)

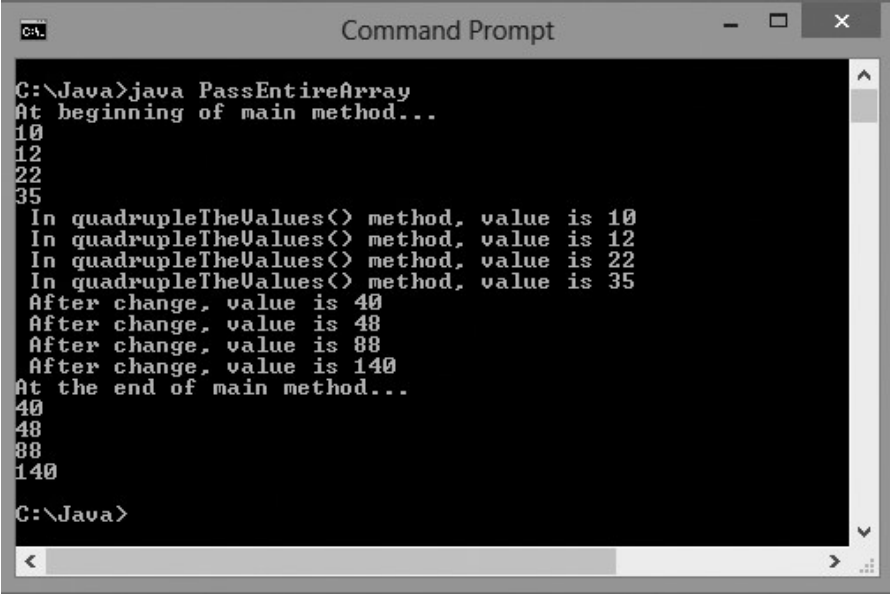
```
29 public static void quadrupleTheValues(int [] vals)
30 {
31     final int LENGTH = 4;
32     int x;
33     x = 0;
34     // Print array values before they are changed
35     while(x < LENGTH)
36     {
37         System.out.println(
38             " In quadrupleTheValues() method, value is " +
39             vals[x]);
40         x++;
41     }
42     x = 0;
43     while(x < LENGTH) // This loop changes array values
44     {
45         vals[x] = vals[x] * 4;
46         x++;
47     }
48     x = 0;
49     // Print array values after they are changed
50     while(x < LENGTH)
51     {
52         System.out.println(" After change, value is " +
53             vals[x]);
54         x++;
55     }
56 } // End of quadrupleTheValues method
57 } // End of PassEntireArray class.
```

Figure 9-9 Java code for the Pass Entire Array program

When an entire array is passed to a method, the square brackets and the size are not included. Note that when you pass an entire array to a method, the array is **passed by reference**. So, instead of a copy of the array being passed, the memory address of the array is passed. This gives the method access to that memory location; the method can then change the values stored in the array if necessary.

Program control is then transferred to the `quadrupleTheValues()` method. The header for the method on line 29 includes one parameter, `int [] vals`. The syntax for declaring an array as a formal parameter includes the parameter's data type, followed by empty square brackets, followed by a local name for the array. Note that a size is not included within the square brackets. In the `quadrupleTheValues()` method, the first `while` loop on lines 35 through 41 prints the values stored in the array, and the second `while` loop on lines 43 through 47 accesses each element in the array, quadruples the value, and then stores the quadrupled values in the array at their same location. The third `while` loop on lines 50 through 55 prints the changed values now stored in the array. Program control is then returned to the location in the `main()` method where the method was called.

When program control returns to the `main()` method, the next statements to execute (lines 19 through 26) are responsible for printing out the values stored in the array once more. The output from this program is displayed in Figure 9-10.



```

C:\Java>java PassEntireArray
At beginning of main method...
10
12
22
35
In quadrupleTheValues() method, value is 10
In quadrupleTheValues() method, value is 12
In quadrupleTheValues() method, value is 22
In quadrupleTheValues() method, value is 35
After change, value is 40
After change, value is 48
After change, value is 88
After change, value is 140
At the end of main method...
40
48
88
140
C:\Java>

```

Figure 9-10 Output from the Pass Entire Array program

As shown in Figure 9-10, the array values printed at the beginning of the `main()` method (lines 12 through 16) are the values with which the array was initialized. Next, the `quadrupleTheValues()` method prints the array values (lines 35 through 41) again before they are changed. The values remain the same as the initialized values. The `quadrupleTheValues()` method then prints the array values again after the values are quadrupled (lines 50 through 55). After the call to `quadrupleTheValues()`, the `main()` method prints the array values one last time (lines 22 through 26). These are the quadrupled values, indicating that the `quadrupleTheValues()` method has access to the memory location where the array is stored and can permanently change the values stored there.

You can also pass a single array element to a method, just as you pass a variable or constant. The following Java code initializes an array named `someNums`, declares a variable named `newNum`, and passes one element of the array to a method named `tripleTheNumber()`.

```

int someNums[] = {10, 12, 22, 35};
int newNum;
newNum = tripleTheNumber(someNums[1]);

```

The following Java code includes the header for the method named `tripleTheNumber()` along with the code that triples the value passed to it.

```
public static int tripleTheNumber(int num)
{
    int result;
    result = num * 3;
    return result;
}
```

Exercise 9-5: Passing Arrays to Methods

In this exercise, you use what you have learned about passing arrays and array elements to methods to answer Questions 1–3.

1. Given the following method calls, write the method's header:
 - a. `int marchBirthdays [] = {3, 12, 13, 22, 27, 30};`
`printBirthdays(marchBirthdays);`

 - b. `double octoberInvoices [] = {100.00, 200.00, 55.00, 230.00};`
`total = monthlyIncome(octoberInvoices);`

 - c. `double balance[] = {34.56, 33.22, 65.77, 89.99};`
`printBill(balance[1]);`

2. Given the following method headers, write a method call:
 - a. `public static void midtermGrades(String [] name, double [] grades)`

 - b. `public static int printAverage(int [] nums)`

3. Given the following method header (in which `sal` is one element of an array of `doubles`), write a method call:
 - a. `public static void bonus(double sal)`

Lab 9-5: Passing Arrays to Methods

In this lab, you complete a partially written Java program that reverses the order of five numbers stored in an array. The program should first print the five numbers stored in the array. Next, the program passes the array to a method where the numbers are reversed. Finally, the main program should print the reversed numbers.

The source code file provided for this lab includes the necessary variable declarations. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `Reverse.java` using Notepad or the text editor of your choice.
2. Write the Java statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file, `Reverse.java`.
5. Execute the program.

Overloading Methods

You can **overload** methods by giving the same name to more than one method. Overloading methods is useful when you need to perform the same action on different types of inputs. For example, you may want to write multiple versions of an `add()` method—one that can add two integers, another that can add two `doubles`, another that can add three integers, and another that can add two integers and a `double`. Overloaded methods have the same name, but they must either have a different number of arguments or the arguments must be of a different data type. Java figures out which method to call based on the method's name and its arguments, the combination of which is known as the method's **signature**. The signature of an overloaded method consists of the method's name and its argument list; it does not include the method's return type.

Overloading methods allows a Java programmer to choose a meaningful name for a method and also permits the use of polymorphic code. **Polymorphic** code is code that acts appropriately depending on the context. (The word polymorphic is derived from the Greek words *poly*, meaning "many," and *morph*, meaning "form.") Polymorphic methods in Java can take many forms. You will learn more about polymorphism in other Java courses, when you learn more about object-oriented programming. For now, you can use overloading to write methods that perform the same task but with different data types.

In Chapter 9 of *Programming Logic and Design*, you studied the design for an overloaded method named `printBill()`. One version of the method includes a numeric parameter, a second version includes two numeric parameters, a third version includes a numeric parameter and a `String` parameter, and a fourth version includes two numeric parameters and a `String` parameter. All versions of the `printBill()` method have the same name with a different signature; therefore, it is an overloaded method. In Figure 9-11 you see a Java program that includes the four versions of the `printBill()` method.

```
1 // Overloaded.java - This program illustrates overloaded
2 // methods.
3 // Input: None.
4 // Output: Bill printed in various ways.
5 import javax.swing.*;
6 public class Overloaded
7 {
8     public static void main(String args[])
9     {
10         double bal = 250.00, discountRate = .05;
11         String msg = "Due in 10 days.";
12         printBill(bal); // Call version #1.
13         printBill(bal, discountRate); // Call version #2.
14         printBill(bal, msg); // Call version #3.
15         printBill(bal, discountRate, msg); // Call version #4.
16         System.exit(0);
17     } // End of main() method.
18 }
```

Figure 9-11 Program that uses overloaded `printBill()` methods (*continues*)

(continued)

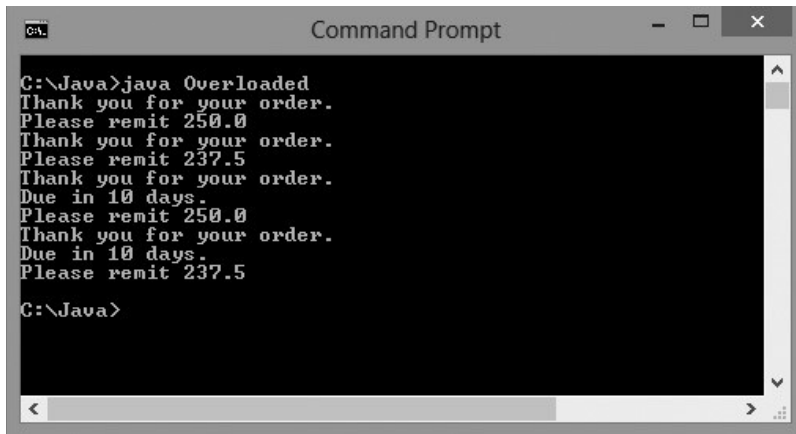
```
19 // printBill() method version #1.
20 public static void printBill(double balance)
21 {
22     System.out.println("Thank you for your order.");
23     System.out.println("Please remit " + balance);
24 } // End of printBill version #1 method.
25
26 // printBill() method version #2.
27 public static void printBill(double balance,
28                             double discount)
29 {
30     double newBal;
31     newBal = balance - (balance * discount);
32     System.out.println("Thank you for your order.");
33     System.out.println("Please remit " + newBal);
34 } // End of printBill version #2 method.
35
36 // printBill() method version #3.
37 public static void printBill(double balance,
38                             String message)
39 {
40     System.out.println("Thank you for your order.");
41     System.out.println(message);
42     System.out.println("Please remit " + balance);
43 } // End of printBill version #3 method.
44
45 // printBill() method version #4.
46 public static void printBill(double balance,
47                             double discount,
48                             String message)
49 {
50     double newBal;
51     newBal = balance - (balance * discount);
52     System.out.println("Thank you for your order.");
53     System.out.println(message);
54     System.out.println("Please remit " + newBal);
55 } // End of printBill version #4 method.
56 } // End of Overloaded class.
```

Figure 9-11 Program that uses overloaded `printBill()` methods

On line 12, the first call to the `printBill()` method passes one argument, the variable named `bal`, which is declared as a `double`. This causes the runtime system to find and execute the `printBill()` method that is written to accept one `double` as an argument (line 20). The third call to the `printBill()` method (line 14) passes two arguments, a `double` and a `String`. This causes the runtime system to find and execute the `printBill()` method that is written to accept a `double` and a `String` as arguments (line 37). You can compile and execute this program if you would like to verify that a different version of the `printBill()` method is called when a different number of arguments are passed or arguments of different data types

are passed. The program, named `Overloaded.java`, is included with the data files provided with this book. The output generated by this program is shown in Figure 9-12.

170



```
C:\Java>java Overloaded
Thank you for your order.
Please remit 250.0
Thank you for your order.
Please remit 237.5
Due in 10 days.
Please remit 250.0
Thank you for your order.
Due in 10 days.
Please remit 237.5
C:\Java>
```

Figure 9-12 Output from the `Overloaded` program

Exercise 9-6: Overloading Methods

In this exercise, you use what you have learned about overloading methods to answer Question 1.

```
1 // Method headers
2 public static int sum(int num1, int num2)
3 public static int sum(int num2, int num2, int num3)
4 public static double sum(double num1, double num2)
5 double number1 = 1.0, ans1;
6 int number2 = 5, ans2;
```

Figure 9-13 Method headers for Exercise 9-6

1. In Figure 9-13, which method header would the following method calls match? Use a line number as your answer.
 - a. `ans2 = sum(2, 7, 9);`

 - b. `ans1 = sum(number2, number2);`

 - c. `ans1 = sum(10.0, 7.0);`

```
d. ans2 = sum(3, 5);
```

```
e. ans2 = sum(2, 8, number1);
```

Lab 9-6: Overloading Methods

In this lab, you complete a partially written Java program that computes hotel guest rates at Cornwall's Country Inn. The program is described in Chapter 9, Exercise 11, in *Programming Logic and Design*. In this program, you should include two overloaded methods named `computeRate()`. One version accepts a number of days and calculates the rate at \$99.99 per day. The other accepts a number of days and a code for a meal plan. If the code is *A*, three meals per day are included, and the price is \$169.00 per day. If the code is *C*, breakfast is included, and the price is \$112.00 per day. All other codes are invalid. Each method returns the rate to the calling program where it is displayed. The main program asks the user for the number of days in a stay and whether meals should be included; then, based on the user's response, the program either calls the first method or prompts for a meal plan code and calls the second method. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `Cornwall.java` using Notepad or the text editor of your choice.
2. Write the Java statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file, `Cornwall.java`.
5. Execute the program.

Using Java's Built-in Methods

Throughout this book, you have used some of Java's built-in methods, such as the `println()` method, the `showInputDialog()` method, and the `parseInt()` and `parseDouble()` methods. In this section, you will look at another built-in method named `format()` that allows you to control the number of places displayed after the decimal point when you print a value of data type `double`. Using the `format()` method is one of several ways to control the number of places displayed after a decimal point.

In the code sample that follows, you see that the `format()` method expects two arguments, a `String` constant and a value to format. Notice the value to format is a variable named `valToFormat` that is declared as data type `double`.

```
double valToFormat = 1234.12;  
System.out.format("%.3f%n", valToFormat);
```

In the preceding code sample, the `String` constant, `%.3f%n`, is a format **specifier** that describes how the value should be formatted. Format specifiers begin with a percent sign (`%`) and end with a converter. The **converter** is a character indicating the type of argument to be formatted. In this example, the `f` in `%.3f` specifies that the value to be formatted is a floating-point value. In between the percent sign (`%`) and the converter, you can include optional flags and specifiers. In this example, `.3` is an optional flag that specifies that you want to display three places after the decimal point. The format specifier, `%n`, indicates that a newline character should be displayed. The output from this code sample is `1234.120`.

As you continue to learn more about Java, you will be introduced to many more built-in methods that you can use in your programs.

Exercise 9-7: Using Java's Built-in Methods

In this exercise, you use the online documentation supplied by Oracle to answer Questions 1–8. Go to <http://download.oracle.com/javase/6/docs/api/> to access Java's online documentation. Scroll down until you see the word *String* in the left pane, under *All Classes*. Click *String* to access the documentation regarding Java's `String` class. Read the information about the built-in methods that belong to the `String` class, and then answer the following questions:

1. What does the `concat()` method do?

2. What data type does the `concat()` method return?

3. What does the `isEmpty()` method do?

4. What data type does the `isEmpty()` method return?

5. How many arguments does the `charAt()` method require?

6. What is the data type of the argument(s)?

7. Is the `startsWith()` method overloaded?

8. How many versions of the `startsWith()` method are listed?

Lab 9-7: Using Java's Built-in Methods

In this lab, you complete a partially written Java program that includes built-in methods that convert `String`s to all uppercase or all lowercase. The program prompts the user to enter any `String`. To end the program, the user can enter `done`. For each `String` entered, call the built-in methods `toLowerCase()` and `toUpperCase()`. The program should call these methods using a `String` object, followed by a dot (`.`), followed by the name of the method. Both of these methods return a `String` with the `String` changed to uppercase or lowercase. Here is an example:

```
String sample = "This is a String.";
String result;
result = sample.toLowerCase();
result = sample.toUpperCase();
```

The source code file provided for this lab includes the necessary variable declarations and the necessary input and output statements. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `ChangeCase.java` using Notepad or the text editor of your choice.
2. Write the Java statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file, `ChangeCase.java`.
5. Execute the program.