APPENDIX C

Formatting Output

In this appendix, you will:

- Round numbers
- Use the `printf()` method
- Use the `DecimalFormat` class

Rounding Numbers

In Chapter 2 and Appendix B, you learned about the imprecision of floating-point numbers. For example, if you write a program that subtracts 2.00 from 2.20, the result is not 0.20—it is 0.20000000000000018. To eliminate odd-looking output and nonintuitive comparisons caused by imprecise calculations in floating-point numbers, you can take the approach shown in the class in Figure C-1. If you want to round a number to two decimal places, note the shaded steps in the figure:

- Multiply the value by 100. So, for example, 0.20000000000000018 becomes 20.000000000000018.
- Add 0.5. This increases a value's whole number part by 1 if the fractional part is 0.5 or greater. For example, 41.6 would become 42.1. In this case, 20.000000000000018 becomes 20.500000000000018.
- Cast the value to an integer. In this case, 20.500000000000018 becomes 20.
- Divide by 100. In this case, the value becomes 0.20.

```
public class RoundingDemo1
{
   public static void main(String[] args)
   {
       double answer = 2.20 - 2.00;
       boolean isEqual;
       isEqual = answer == 0.20;
       System.out.println("Before conversion");
       System.out.println("answer is " + answer);
       System.out.println("isEqual is " + isEqual);
       answer = answer * 100;
       answer = answer + 0.5;
       answer = (int) answer;
       answer = answer / 100;
       isEqual = answer == 0.20;
       System.out.println("After conversion");
       System.out.println("answer is " + answer);
       System.out.println("isEqual is " + isEqual);
   }
}
```

Figure C-1 The `RoundingDemo1` class

Figure C-2 shows the output of the program. Without rounding, the displayed difference between 2.20 and 2.00 is 0.2000000000000000018. However, after applying the rounding technique, the result is displayed as 0.2 as expected.

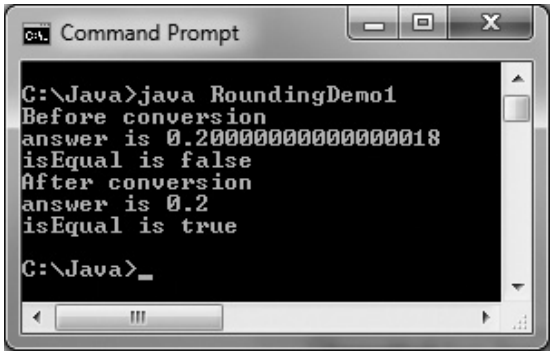

Figure C-2 Output of the RoundingDemo1 program

As an alternative, you can use the round() method that is supplied with Java's Math class. The round() method returns the nearest long value. Figure C-3 shows a program that multiplies the double answer by 100, rounds it, and then divides by 100.0. The output is identical to that shown in Figure C-2.

```
public class RoundingDemo2
{
   public static void main(String[] args)
   {
        double answer = 2.20 - 2.00;
        boolean isEqual;
        isEqual = answer == 0.20;
        System.out.println("Before conversion");
        System.out.println("answer is " + answer);
        System.out.println("isEqual is " + isEqual);
        answer = answer * 100;
        long roundedAnswer = Math.round(answer);
        answer = roundedAnswer / 100.0;
        isEqual = answer == 0.20;
        System.out.println("After conversion");
        System.out.println("answer is " + answer);
        System.out.println("isEqual is " + isEqual);
   }
}
```

Figure C-3 The RoundingDemo2 class

Using the printf() Method

When you display numbers using the println() method in Java applications, it sometimes is difficult to make numeric values appear as you want. For example, in the output in Figure C-2, the difference between 2.20 and 2.00 is displayed as 0.2. By default, Java eliminates

trailing zeros when floating-point numbers are displayed because they do not add any mathematical information. You might prefer to see 0.20 because the original numbers were both expressed to two decimal places, or, in particular, if the values represent currency.

Additionally, you frequently want to align columns of numeric values. For example, Figure C-4 shows a `NumberList` application that contains an array of floating-point values. The application displays the values using a `for` loop, but as the output in Figure C-5 shows, the numbers are not aligned by the decimal point as you usually would want numbers to be aligned. Because the `println()` method displays values as `String`s, the displayed values are left-aligned, just as series of words would be. The numeric values are accurate; they just are not attractively arranged.

```
public class NumberList
{
   public static void main(String[] args)
   {
      double[] list = {0.20, 2.00, 2.20, 22.22,
         22.20, 222.00, 222.22};
      int x;
      for(x = 0; x < list.length; ++x)
         System.out.println(list[x]);
   }
}
```

Figure C-4 The `NumberList` application

Figure C-5 Output of the `NumberList` application

The **`System.out.printf()` method** is used to format numeric values. It is a newer Java feature that was first included in the `Formatter` class in Java 1.5.0. (This is the internal version number of the Java Development Kit; the external version number is 5.0.) Because this class is contained in the `java.util` package, you do not need to include any `import` statements to use it. The `printf()` method allows you to format numeric values in two useful ways:

- By specifying the number of decimal places to display
- By specifying the field size in which to display values

The `Formatter` class contains many formats that are not covered here. To view the details of formatting data types such as `BigDecimal` and `Calendar`, visit the Java Web site.

C programmers use a `printf()` function that is very similar to Java's `printf()` method. Although the `printf()` method is used in these examples, in Java, you can substitute `System.out.format()` for `System.out.printf()`. There is no difference in the way you use these two methods.

When creating numeric output, you can specify a number of decimal places to display by using the `printf()` method with two types of arguments that represent the following:

- A format string
- A list of arguments

A **format string** is a string of characters; it includes optional text (that is displayed literally) and one or more format specifiers. A **format specifier** is a placeholder for a numeric value. Within a call to `printf()`, you include one argument (either a variable or a constant) for each format specifier.

The format specifiers for general, character, and numeric types contain the following elements, in order:

- A percent sign (%), which indicates the start of every format specifier
- An optional argument index, which is an integer indicating the position of the argument in the argument list. The integer is followed by a dollar sign. You will learn more about this option later in this appendix.
- Optional flags that modify the output format. The set of valid flags depends on the data type you are formatting. You can find more details about this feature at the Java Web site.
- An optional field width, which is an integer indicating the minimum number of characters to be written to the output. You will learn more about this option later in this appendix.
- An optional precision factor, which is a decimal point followed by a number and typically used to control the number of decimal places displayed. You will learn more about this option in the next section.
- The required conversion character, which indicates how its corresponding argument should be formatted. Java supports a variety of conversion characters, but the three you want to use most frequently are `d`, `f`, and `s`, the characters that represent decimal (base 10 integer), floating-point (`float` and `double`), and string values, respectively.

Other conversion characters include those used to display hexadecimal numbers and scientific notation. If you need these display formats, you can find more details at the Java Web site.

For example, you can use the `ConversionCharacterExamples` class in Figure C-6 to display a declared integer and `double`. The `main()` method of the class contains three `printf()` statements. The three calls to `printf()` in this class each contain a format string; the first two calls contain a single additional argument, and the last `printf()` statement contains two arguments after the string. None of the format specifiers in this class use any of the optional parameters—only the required percent sign and conversion character. The first `printf()` statement uses `%d` in its format string as a placeholder for the integer argument at the end. The second `printf()` statement uses `%f` as a placeholder for the floating-point argument at the end. The last `printf()` statement uses both a `%d` and `%f` to indicate the positions of the integer and floating-point values at the end, respectively. If you attempt to use a conversion character that is invalid for the data type, the program will compile, but it will throw an exception during execution when it encounters the wrong conversion character for the value being displayed.

```
public class ConversionCharacterExamples
{
   public static void main(String[] args)
   {
      int age = 23;
      double money = 123.45;
      System.out.printf("Age is %d\n",age);
      System.out.printf("Money is $%f\n", money);
      System.out.printf
         ("Age is %d and money is $%f\n", age, money);
   }
}
```

Figure C-6 The `ConversionCharacterExamples` application

Figure C-7 shows the output of the program, in which the values are inserted in the appropriate places in their strings. Note that floating-point values are displayed with six decimal positions by default.

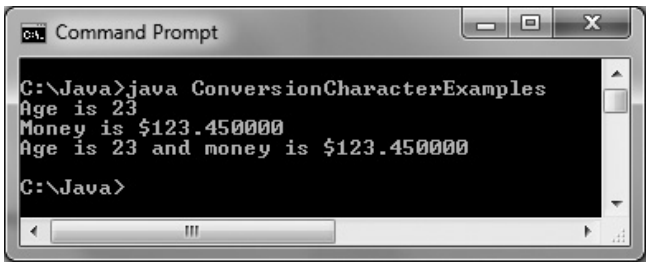

Figure C-7 Output of the `ConversionCharacterExamples` application

Notice that in the `ConversionCharacterExamples` class, the output appears on three separate lines only because the newline character ('\n') has been included at the end of each `printf()` format string. Unlike the `println()` statement, `printf()` does not include an automatic new line.

Specifying a Number of Decimal Places to Display with printf()

You can control the number of decimal places displayed when you use a floating-point value in a printf() statement by adding the optional precision factor to the format specifier. Between the percent sign and the conversion character, you can add a decimal point and the number of decimal positions to display. For example, the following statements produce the output "Money is $123.45", displaying the money value with just two decimal places instead of six, which would occur without the precision factor:

```
double money = 123.45;
System.out.printf("Money is $%.2f\n", money);
```

Similarly, the following statements display 8.10. If you use the println() equivalent with amount, only 8.1 is displayed; if you use a printf() statement without inserting the .2 precision factor, 8.100000 is displayed.

```
double amount = 8.1;
System.out.printf("%.2f",amount);
```

When you use a precision factor on a value that contains more decimal positions than you want to display, the result is rounded. For example, the following statements produce 100.457 (not 100.456), displaying three decimals because of the precision factor.

```
double value = 100.45678;
System.out.printf("%.3f",value);
```

You cannot use the precision factor with an integer value; if you do, your program will throw an IllegalFormatConversionException.

Specifying a Field Size with printf()

You can indicate a field size in which to display output by using an optional integer as the field width. For example, the NumberList2 class in Figure C-8 displays each array element in a field with a size of 6, using two decimal places. Figure C-9 shows the output of the application. Each value is displayed right-aligned in its field; for example, 0.20 is preceded by two blank spaces, and 22.20 is preceded by one blank space. If a numeric value contains more positions than you indicate for its printf() field size, the field size is ignored, and the entire value is displayed.

```
public class NumberList2
{
   public static void main(String[] args)
   {
      double[] list = {0.20, 2.00, 2.20, 22.22,
         22.20, 222.00, 222.22};
      int x;
      for(x = 0; x < list.length; ++x)
         System.out.printf("%6.2f\n", list[x]);
   }
}
```

Figure C-8 The NumberList2 class

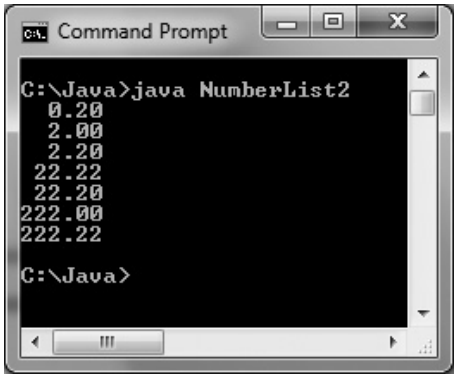

Figure C-9 Output of the `NumberList2` class

Throughout this book, you have been encouraged to use named constants for numeric values instead of literal constants, so that your programs are clearer. In the program in Figure C-9, you could define constants such as:

```
final int DISPLAY_WIDTH = 6;
final int DISPLAY_DECIMALS = 2;
```

Then the `printf()` statement would be:

```
System.out.printf("%" + DISPLAY_WIDTH + "." +
   DISPLAY_DECIMALS + "f\n", list[x]);
```

Another, perhaps clearer alternative is to define a format string such as the following:

```
final String FORMAT = "%6.2f\n";
```

Then the `printf()` statement would be:

```
System.out.printf(FORMAT, list[x]);
```

You can specify that a value be left-aligned in a field instead of right-aligned by inserting a negative sign in front of the width. Although you can do this with numbers, most often you choose to left-align strings. For example, the following code displays five spaces followed by "hello" and then five spaces followed by "there". Each string is left-aligned in a field with a size of 10.

```
String string1 = "hello";
String string2 = "there";
System.out.printf("%-10s%-10s", string1, string2);
```

Using the Optional Argument Index with `printf()`

The **argument index** is an integer that indicates the position of an argument in the argument list of a `printf()` statement. To separate it from other formatting options, the argument index is followed by a dollar sign ($). The first argument is referenced by `"1$"`, the second by `"2$"`, and so on.

For example, the `printf()` statement in the following code contains four format specifiers but only two variables in the argument list:

```
int x = 56;
double y = 78.9;
System.out.printf("%1$6d%2$6.2f%1$6d%2$6.2f", x, y);
```

The `printf()` statement displays the value of the first argument, `x`, in a field with a size of 6, and then it displays the second argument, `y`, in a field with a size of 6 with two decimal places. Then, the value of `x` is displayed again, followed by the value of `y`. The output appears as follows:

```
56 78.90    56 78.90
```

Using the DecimalFormat Class

The **`DecimalFormat` class** provides ways to easily convert numbers into strings, allowing you to control the display of leading and trailing zeros, prefixes and suffixes, grouping (thousands) separators, and the decimal separator. You specify the formatting properties of `DecimalFormat` with a pattern `String`. The **pattern `String`** is composed of symbols that determine what the formatted number looks like; it is passed to the `DecimalFormat` class constructor.

The symbols you can use in a pattern `String` include:

- A pound sign (#), which represents a digit
- A period (.), which represents a decimal point
- A comma (,), which represents a thousands separator
- A zero (0), which represents leading and trailing zeros when it replaces the pound sign

The pound sign is typed using Shift+3 on standard computer keyboards. It also is called an **octothorpe**, a number sign, a hash sign, square, tic-tac-toe, gate, and crunch.

For example, the following lines of code result in `value` being displayed as 12,345,678.90.

```
double value = 12345678.9;
DecimalFormat aFormat = new DecimalFormat("#,###,###,###.00");
System.out.printf("%s\n", aFormat.format(value));
```

A `DecimalFormat` object is created using the pattern #,###,###,###.00. When the object's `format()` method is used in the `printf()` statement, the first two pound signs and the comma between them are not used because `value` is not large enough to require those positions. The value is displayed with commas inserted where needed, and the decimal portion is displayed with a trailing *0* because the *0*s at the end of the pattern indicate that they should be used to fill out the number to two places.

When you use the `DecimalFormat` class, you must use the import statement `import java.text.*;`. Figure C-10 shows a class that creates a `String` pattern that it passes to the `DecimalFormat` constructor to create a `moneyFormat` object. The class displays an array of values, each in a field that is 10 characters wide. Some of the values require commas, and some do not. Figure C-11 shows the output.

```
import java.text.*;
public class DecimalFormatTest
{
   public static void main(String[] args)
   {
      String pattern = "###,###.00";
      DecimalFormat moneyFormat = new DecimalFormat(pattern);
      double[] list = {1.1, 23.23, 456.249, 7890.1, 987.5678, 65.0};
      int x;
      for(x = 0; x < list.length; ++x)
         System.out.printf("%10s\n", moneyFormat.format(list[x]));
   }
}
```

Figure C-10 The `DecimalFormatTest` class

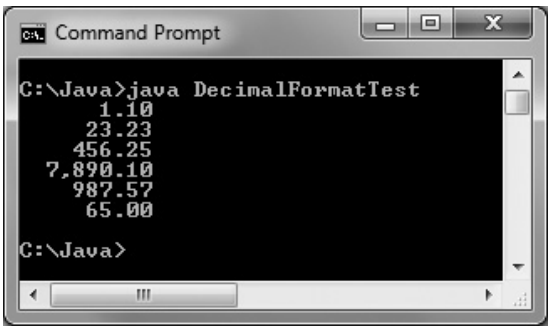

Figure C-11 Output of the `DecimalFormatTest` program

Key Terms

The **`System.out.printf()` method** is used to format numeric values.

A **format string** in a `printf()` statement is a string of characters; it includes optional text (that is displayed literally) and one or more format specifiers.

A **format specifier** in a `printf()` statement is a placeholder for a numeric value.

The **argument index** in a `printf()` statement is an integer that indicates the position of an argument in the argument list.

The **`DecimalFormat` class** provides ways to easily convert numbers into strings, allowing you to control the display of leading and trailing zeros, prefixes and suffixes, grouping (thousands) separators, and the decimal separator.

A **pattern `String`** is composed of symbols that determine what a formatted number looks like; it is passed to the `DecimalFormat` class constructor.

An **octothorpe** is a pound sign.