CHAPTER 8

IMPROVING THE USER INTERFACE

OBJECTIVES

Upon completion of this chapter, you should be able to:

- Construct a query-driven terminal interface.
- Construct a menu-driven terminal interface.
- Construct a graphical user interface.
- Format text, including numbers, for output.
- Handle number format exceptions during input.

Estimated Time: 3.5 hours

VOCABULARY

Application
Controller pattern
Data model
Event-driven
Format flag
Format specifier
Menu-driven program
Model view
Query-controlled input

We do not judge a book by its cover because we are interested in its contents, not its appearance. However, we do judge a software product by its user interface because we have no other way to access its functionality. In this chapter, we explore several ways to improve a program's user interface. First, we present some standard techniques for enhancing terminal-based interfaces, including query-driven input/output (I/O) and menu-driven I/O. This chapter then shows how to format numerical data and examines the output of columns of strings and numbers. The handling of errors in the format of input data is also discussed. Finally, we include a discussion of procedural decomposition and top-down development as methods of organizing a complex task.

8.1 A Thermometer Class

Several examples in this chapter involve converting temperatures between Fahrenheit and Celsius. To support these conversions we first introduce a `Thermometer` class. This class stores the temperature internally in Celsius; however, the temperature can be set and retrieved in either Fahrenheit or Celsius. Here is the code:

```
public class Thermometer {

   private double degreesCelsius;

   public void setCelsius(double degrees){
      degreesCelsius = degrees;
   }
```

```
    public void setFahrenheit(double degrees){
       degreesCelsius = (degrees - 32.0) * 5.0 / 9.0;
    }

    public double getCelsius(){
       return degreesCelsius;
    }

    public double getFahrenheit(){
       return degreesCelsius * 9.0 / 5.0 + 32.0;
    }
}
```

8.2 Repeating Sets of Inputs

In Chapter 4, we introduced two techniques for handling repeating sets of inputs. We called these count-controlled and sentinel-controlled input. We now present a third technique that we call ***query-controlled input***. Before each set of inputs after the first, the program asks the user if there are more inputs. Figure 8-1 shows an example of query-controlled input.

FIGURE 8-1
Interface for a query-controlled temperature conversion program

```
Enter degrees Fahrenheit: 32
The equivalent in Celsius is 0.0

Do it again (y/n)? y

Enter degrees Fahrenheit: 212
The equivalent in Celsius is 100.0

Do it again (y/n)?
```

The program is implemented by means of two classes—a class to handle the user interface and the `Thermometer` class. Following is pseudocode for the interface class:

```
instantiate a thermometer
String doItAgain = "y"
while (doItAgain equals "y" or "Y"){
   read degrees Fahrenheit and set the thermometer
   ask the thermometer for the degrees in Celsius and display
   read doItAgain                  //The user responds with y or n
}
```

The key to this pseudocode is the `String` variable `doItAgain`. This variable controls how many times the loop repeats. Initially, the variable equals `"y"`. As soon as the user enters a string other than `"y"` or `"Y"`, the program terminates. Following is a complete listing of the user interface class:

```
/* Example 8.1: ConvertWithQuery.java
Repeatedly convert from Fahrenheit to Celsius until the user
signals the end.
*/

import java.util.Scanner;

public class ConvertWithQuery {

   public static void main(String [] args){
      Scanner reader = new Scanner(System.in);
      Thermometer thermo = new Thermometer();
      String doItAgain = "y";

      while (doItAgain.equals("y") || doItAgain.equals("Y")){
         System.out.print("\nEnter degrees Fahrenheit: ");
         thermo.setFahrenheit(reader.nextDouble());
         // Consume the trailing newline
         reader.nextLine();
         System.out.println("The equivalent in Celsius is " +
                            thermo.getCelsius());
         System.out.print("\nDo it again (y/n)? ");
         doItAgain = reader.nextLine();
      }
   }
}
```

Note that `"Y"` and `"y"` are not equal. Note also that the input of a `double` leaves behind a newline character in the input stream that must be consumed before the next query is read.

EXERCISE 8.2

1. Describe the structure of a query-controlled loop that processes repeated sets of inputs.
2. Browse into the `String` class in Sun's Java API documentation. Explain how the method `toLowerCase()` could be used to simplify the loop control condition of the query-driven loop in the example program.

8.3 A Menu-Driven Conversion Program

Menu-driven programs begin by displaying a list of options from which the user selects one. The program then prompts for additional inputs related to that option and performs the needed computations, after which it displays the menu again. Figure 8-2 shows how this idea can

be used to extend the temperature conversion program.

FIGURE 8-2
Interface for a menu-driven version of the temperature conversion program

```
1) Convert from Fahrenheit to Celsius
2) Convert from Celsius to Fahrenheit
3) Quit
Enter your option: 1

Enter degrees Fahrenheit: 212
The equivalent in Celsius is 100

1) Convert from Fahrenheit to Celsius
2) Convert from Celsius to Fahrenheit
3) Quit
Enter your option: 2

Enter degrees Celsius: 0
The equivalent in Fahrenheit is 32

1) Convert from Fahrenheit to Celsius
2) Convert from Celsius to Fahrenheit
3) Quit
Enter your option: 3

Goodbye!
```

Here is a pseudocode design, followed by the program:

```
instantiate a thermometer
menuOption = 4
while (menuOption != 3){
 print menu
 read menuOption
 if (menuOption == 1){
    read fahrenheit and set the thermometer
    ask the thermometer to convert and print the results
 }
 else if (menuOption == 2){
    read celsius and set the thermometer
    ask the thermometer to convert and print the results
 }
 else if (menuOption == 3)
    print "Goodbye!"
 else
    print "Invalid option"
}
```

```
/* Example 8.2: ConvertWithMenu.java
A menu-driven temperature conversion program that converts from
Fahrenheit to Celsius and vice versa.
*/
```

```
import java.util.Scanner;

public class ConvertWithMenu {
   public static void main (String [] args) {
      Scanner reader = new Scanner(System.in);
      Thermometer thermo = new Thermometer();
      String menu;           //The multiline menu
      int menuOption;        //The user's menu selection

      //Build the menu string
      menu = "\n1) Convert from Fahrenheit to Celsius"
           + "\n2) Convert from Celsius to Fahrenheit"
           + "\n3) Quit"
           + "\nEnter your option: ";

      //Set up the menu loop
      menuOption = 4;
      while (menuOption != 3){

         //Display the menu and get the user's option
         System.out.print(menu);
         menuOption = reader.nextInt();
         System.out.println("");

         //Determine which menu option has been selected

         if (menuOption == 1){

            //Convert from Fahrenheit to Celsius
            System.out.print("Enter degrees Fahrenheit: ");
            thermo.setFahrenheit(reader.nextDouble());
            System.out.println("The equivalent in Celsius is " +
                               thermo.getCelsius());

         }else if (menuOption == 2){

            //Convert from Celsius to Fahrenheit
            System.out.print("Enter degrees Celsius: ");
            thermo.setCelsius(reader.nextDouble());
            System.out.println("The equivalent in Fahrenheit is " +
                               thermo.getFahrenheit());

         }else if (menuOption == 3)

            //User quits, sign off
            System.out.println("Goodbye!");

          else

            //Invalid option
            System.out.println("Invalid option");
      }
   }
}
```

EXERCISE 8.3

1. What role does a menu play in a program?
2. Describe the structure of a menu-driven command loop.
3. Write the code for a menu that gives the user options to specify the size of a pizza, various toppings (mushrooms, peppers, sausage, etc.), place the order, or return to the main menu.

8.4 Formatted Output with `printf` and `format`

The example programs thus far in this book have used the methods `print` and `println` to output data to the terminal. These methods are easy to use and work well in many applications. However, occasionally a program must format data, especially numbers, more carefully. In this section, we examine a couple of methods to accomplish that.

Using `printf` to Format Numbers

When we talk about the precision of a floating-point number, we are referring to the number of digits to the right of the decimal point supported by the programming language. Java supports the type `double` for numbers with many digits of precision. The methods `print` and `println` do not generally display all of these digits; instead they display only enough to convey the necessary information about the number. Sometimes we get too many digits, and other times we get too few. For example, a whole number would be displayed with a single zero to the right of the decimal point, but in financial applications we would want to see dollars and cents displayed with two zeroes. For example, consider the following code segment:

```
double dollars = 25;
double tax = dollars * 0.125;
System.out.println("Income: $" + dollars);
System.out.println("Tax owed: $" + tax);
```

Here is the output for the preceding code segment:

```
Income: $25.0
Tax owed: $3.125
```

The amount of money on the first line has too few digits to the right of the decimal point, whereas the amount on the second line has too many.

Fortunately, Java includes the method `printf` for formatting output. The following code segment modifies the previous one to show these capabilities for floating-point numbers:

```
double dollars = 25;
double tax = dollars * 0.125;
System.out.printf("Income: $%.2f%n", dollars);
System.out.printf("Tax owed: $%.2f%n", tax);
```

The output produced by this code segment is

```
Income: $25.00
Tax owed: $3.13
```

The parameters of the method `printf` consist of a *format string* and one or more data values, according to the following general form:

```
printf(<format string>, <expression-1>, …, <expression-n>)
```

The format string is a combination of literal string information and formatting information. The formatting information consists of one or more ***format specifiers***. These codes begin with a '%' character and end with a letter that indicates the format type. In our example code segment, the format specifier `%.2f` says display a number in fixed-point format with exactly two digits to the right of the decimal point. The format specifier `%n` says display an end-of-line character. Table 8-1 lists some commonly used format types.

TABLE 8-1
Commonly used format types

CODE	FORMAT TYPE	EXAMPLE VALUE
d	Decimal integer	34
x	Hexadecimal integer	A6
o	Octal integer	47
f	Fixed floating-point	3.14
e	Exponential floating-point	1.67e+2
g	General floating-point (large numbers in exponential and small numbers in fixed-point)	3.14
s	String	Income:
n	Platform-independent end of line	

The symbol `%n` can be used to embed an end-of-line character in a format string. The symbol `%%` (consecutive percent symbols) produces the literal `'%'` character. Otherwise, when the compiler encounters a format specifier in a format string, it attempts to match that specifier to an expression following the string. The two must match in type and position. For example, the code

```
int idNum = 758;
double wage = 10.5;
printf("The wage for employee %d is $%.2f per hour.", idNum, wage);
```

matches the code `%d` to the value of the variable `idNum` and the code `%.2f` to the value of the variable `wage` to produce the output

```
The wage for employee 758 is $10.50 per hour.
```

Text Justification and Multiple Columns

Data-processing applications frequently display tables that contain columns of words and numbers. Unless these tables are formatted carefully, they are unreadable. To illustrate, Figure 8-3 shows a table of names, sales, and commissions, with and without formatting. Although both tables contain exactly the same data, only the formatted one is readable. The key feature of a formatted table is that each column has a designated width, and all the values in a column are justified in the same manner—either to the left, right, or center of the column. In the sales table, the names in the first column are left-justified, and the numbers in the other two columns are right-justified.

FIGURE 8-3
A table of sales figures shown with and without formatting

```
NAME SALES COMMISSION
Catherine 23415 2341.5
Ken 321.5 32.15
Martin 4384.75 438.48
Tess 3595.74 359.57
```

Version 1: Unreadable without formatting

```
NAME                   SALES      COMMISSION
Catherine          23415.00         2341.50
Ken                  321.50           32.15
Martin              4384.75          438.48
Tess                3595.74          359.57
```

Version 2: Readable with formatting

The columns of text in Version 2 are produced by displaying pieces of text that are justified within fields. A field consists of a fixed number of columns within which the characters of a data value can be placed. A data value is left-justified when its display begins in the leftmost column of its field. The value is right-justified when its display ends in the rightmost column of its field. In either case, trailing or leading spaces are used to occupy columns that are not filled by the value. The column of names in Version 2 is left-justified, whereas the two columns of numbers are both right-justified.

The method `printf` includes several ***format flags*** that support the justification of text as well as other format styles. Some commonly used format flags are listed in Table 8-2.

TABLE 8-2
Some commonly used format flags

FLAG	WHAT IT DOES	EXAMPLE VALUE
-	Left justification	`34`
,	Show decimal separators	`20,345,000`
0	Show leading zeroes	`002.67`
^	Convert letters to uppercase	`1.56E+3`

To output data in formatted columns, we establish the width of each field and then choose the appropriate format flags and format specifiers to use with `printf`. The width of a field that contains a `double` appears before the decimal point in the format specifier `f`. For example, the format specifer `%6.2f` says to use a precision of 2 digits and a field of 6 columns when displaying the floating-point number. One column is reserved for the decimal point and the number is right-justified by default. Thus, the number 4.5 is displayed with three leading spaces. The number 45000.5, on the other hand, exceeds the size of the field and is not right-justified. In general, when the width of the field is less than or equal to the string representation of the number, the string is left-justified. The same rule holds when the field is omitted before the decimal point. You should take care to ensure that the field width for your data is large enough to accommodate them.

The format specifier `%10s` says to use a field of 10 columns to display a string with right justification. However, for the leftmost column of a table, we would like the strings to be left-justified. This is accomplished by using the format flag '–' in the format specifier. Thus, `%-10s` says to use left justification when filling the columns for the string. Table 8-3 shows some examples of format strings and the output results.

TABLE 8-3
Some example format strings and their outputs

VALUES	FORMAT STRING	OUTPUT
34, 56.7	"%d%7.2f"	34 56.70
34, 56.7	"%4d%7.2f"	34 56.70
34, 56.7	"%-4d$%7.2f"	34 $ 56.70
34, 56.7	"%-4d$%.2f"	34 $56.70

Our next program example puts these ideas together to display a formatted table of data contained in two text files. The first file contains the last names of employees. The second file contains their salaries. The program assumes that each file contains the same number of data values and that the position of an employee's name in one file corresponds to the position of her salary in the other file. We also know that no name exceeds 15 letters and no salary exceeds 6 figures. To illustrate the required format for the output, we are given the following example:

```
NAME                          SALARY
Barker                      4,000.00
Lambert                    36,000.00
Osborne                   150,000.00
```

Note that the numbers contain commas in the standard places. Using this information, we present the following design of the program in pseudocode:

```
Open the file of names
Open the file of salaries
Output the formatted header of the table
While there are more names in the file of names
   Read a name from the file of names
   Read a salary from the file of salaries
   Output a formatted line containing the name and salary
```

To format the column of names, we left-justify the header and each name within 16 columns. To format the column of salaries, we right-justify the header within 12 columns and each salary is given 12 columns of width and two columns of precision. In addition, we use a format flag to place commas in the numbers. Here is the code for the program:

```
// Example 8.3: Display a table of names and salaries

import java.io.*;
import java.util.Scanner;

public class DisplayTable{

   public static void main(String[] args) throws IOException{
      Scanner names = new Scanner(new File("names.txt"));
      Scanner salaries = new Scanner(new File("salaries.txt"));
      System.out.printf("%-16s%12s%n", "NAME", "SALARY");
      while (names.hasNext()){
         String name = names.nextLine();
         double salary = salaries.nextDouble();
         System.out.printf("%-16s%,12.2f%n", name, salary);
      }
   }
}
```

Formatting with `String.format`

The `String` method `format` can be used to build a formatted string. This method expects the same parameters as `printf` and returns a formatted string. For example, the call `String.format("Income: %,.2f", 2500000)` returns the string `"Income: 2,500,000.00"`.

EXERCISE 8.4

1. Write code segments to output the following formatted strings:
   - **a.** " One space"
   - **b.** " Two spaces"
   - **c.** "Three spaces "
   - **d.** The value of `int` variable `i`, right-justified in a field of six columns
   - **e.** The value of `double` variable `d`, right-justified in a field of 10 columns with a precision of 2

2. Write the values returned by the following expressions (include spaces within the strings where relevant):
   - **a.** `String.format("%-10s%,10.2f", "Price", 10000.50)`
   - **b.** `String.format("%6d%7d", 45, 632)`
   - **c.** `String.format("%5.2f", 34.543)`

8.5 Handling Number Format Exceptions During Input

In Sections 4.4 and 7.1, we discussed the idea of programs that respond to errors, particularly when input data are invalid. If data are found to be invalid after input, the program can display an error message and prompt for the data again. Typical errors are input numbers that do not lie within a certain range. However, what happens if the user enters a number in an invalid format? For example, the user might type in the characters "12r8" when she really intends "1258". The datum in that case is not even a number, let alone an invalid one.

Clearly, the implementer of the input methods must detect and do something about number format errors. The `Scanner` methods `nextInt` and `nextDouble` do just that. When format errors are detected, these methods throw an exception that halts the program. The bad format is detected, but before the client code can react to the error. At that point, it's too late; the program is halted. This behavior may be acceptable during testing and debugging. However, crashing with an arcane error message is the last thing we would want the final release of a program to do.

Fortunately, there is a way to detect and respond to an exception under program control so that the exception does not halt the program. The programmer embeds the call to an input method in a `try-catch` statement. As its name implies, this statement consists of two parts, as shown in the following simplified form:

```
try{
   <statements that might throw exceptions>
}catch(Exception e){
   <code to recover from an exception if it's thrown>
}
```

The statements within the `try` clause are executed until one of them throws an exception. If that happens, an exception object is created and sent immediately to the `catch` clause. The code within the `catch` clause is then executed. Alternatively, if no statement throws an exception within the `try` clause, the `catch` clause is skipped. There are actually many specific types of exceptions that could be thrown; our example `try-catch` statement catches all of them. A more detailed discussion of Java exceptions appears in Appendix F.

The next program modifies the query-driven example of Section 8.2 to illustrate the recovery from number format errors. We add a nested input loop that cycles around a `try-catch` statement. The display of the prompt, the input statement, and a `break` statement occur within the `try` clause. If no exception occurs during input, the thermometer is set with the input number and the input loop is broken. Otherwise, when an exception occurs, the `catch` clause immediately displays an error message and consumes the trailing end of line, and the input loop continues. Here is the code:

```
/* Example 8.4: ConvertWithQuery.java
Repeatedly convert from Fahrenheit to Celsius until the user
signals the end. Recovers from a number format error with an
error message.
*/

import java.util.Scanner;
```

```
public class ConvertWithQuery {
   public static void main(String [] args) {
      Scanner reader = new Scanner(System.in);
      Thermometer thermo = new Thermometer();
      String doItAgain = "y";

      while (doItAgain.equals("y") || doItAgain.equals("Y")){
         // Nested loop until input number is well-formed
         while (true)
            try{
               // Attempt statements that might throw exceptions
               System.out.print("\nEnter degrees Fahrenheit: ");
               thermo.setFahrenheit(reader.nextDouble());
               break;
            }catch(Exception e){
               // Code for error recovery
               System.out.println("Error in number format!");
               // Consume the trailing newline due to bad input
               reader.nextLine();
            }
         System.out.println("The equivalent in Celsius is " +
               thermo.getCelsius());
         System.out.print("\nDo it again (y/n)? ");
         // Consume the trailing end of line
         reader.nextLine();
         doItAgain = reader.nextLine();
      }
   }
}
```

As you can see, this version of the program is robust for number format errors. The input loop continues until a properly formatted number is entered.

EXERCISE 8.5

1. Assume that a program is trying to read a double value from the keyboard. The user enters the characters "$12.55". Explain what happens next.

2. Explain how a try-catch statement works.

3. Write a code segment that loops until the user has entered a well-formed double value at the keyboard.

8.6 Graphics and GUIs

Because developing GUIs in Java is usually quite complicated, many introductory textbooks either restrict themselves to terminal-based I/O or present a rather limited and distorted version of GUIs. Thus far in this book, we have gradually introduced two limited types of GUIs: those that display and allow the user to interact with graphical images (Chapters 2–7), and those that use

Extra Challenge

This Graphics and GUIs section gives you the opportunity to explore concepts and programming techniques required to develop modern graphics applications and graphical user interfaces. This material is not required in order to proceed with the other chapters of the book.

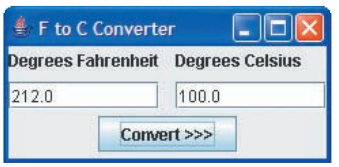
dialog boxes for numeric and text I/O (Chapter 4). Although dialog box I/O suffices for some simple applications, it really does not reflect the true power of a GUI. A program that uses dialog box I/O forces the user to respond to a rigid sequence of pop-up prompts for data. If more than a single data value must be entered, this process can become quite tedious for the user. Moreover, the user might want to refer back to a datum already entered in the sequence before entering the next value, but the earlier dialog box has already disappeared. This problem only gets worse as the number of data values increases. In contrast, a more realistic and powerful GUI presents the user with entry fields for many of the data values simultaneously. Many different command options are also available, via command buttons or drop-down menus. The user can edit any of the fields until she's satisfied that the right combination of the data is present and then select the appropriate command. The results of the computation can then be displayed in the same window.

In this section, we present several realistic but still fairly simple GUIs of this type. In the process, we show how realistic GUIs also allow a programmer to better organize code to solve problems.

The Model/View/Controller Pattern

We start with a very simple application: the temperature conversion program discussed in Section 8.1. Figure 8-4 shows the user interface for a GUI-based version of this program. To use the program, we enter a temperature in the field labeled "Degrees Fahrenheit" and click the Convert>>> button below it. The converted temperature is then displayed in the field labeled "Degrees Celsius." We can repeat the process as many times as desired and click the window's close icon when finished.

FIGURE 8-4
Interface for the GUI-based temperature conversion program



As you might expect, the GUI version can use the same `Thermometer` class used by all of the other versions of the program in this chapter. We call this class the ***data model***, or *model* for short. Its responsibilities are to initialize and manage the data used by the program. A second set of classes is called the *view*. The view consists of a window, buttons, data fields, and labels visible to the user. The view is responsible for displaying a view of the data model and providing the user with visible controls for interaction. A third set of classes is called the *controller*. These controller classes are listeners such as those introduced in Chapters 4 through 7. They are responsible for handling user interaction with the program, usually by responding to events that occur in the view. For example, a listener attached to a button is informed that the user has clicked it. The listener responds by sending a message to the data model and then updates the view with the results.

These three sets of classes make up the ***model/view/controller pattern***, which we use to structure all of the GUIs in this section. Last but not least, we use a separate class that sets up these other elements in a `main` method, so as to provide an entry point for running a Java program. We call this type of class, for lack of a better term, the ***application***.

Putting Together the Temperature Conversion Program

Our program consists of four programmer-defined classes: `ConvertWithGUI`, `GUIWindow`, `FahrenheitListener`, and `Thermometer`. `ConvertWithGUI` is the application class. It defines a `main` method that instantiates the application's main window, sets some of its attributes, and makes it visible. The code for this class is quite straightforward:

```
/* ConvertWithGUI.java
Application class for a GUI-based temperature conversion
program that coverts from Fahrenheit to Celsius.
*/

import javax.swing.*;

public class ConvertWithGUI{

   // Execution begins in the method main as usual.
   public static void main(String[] args){
      GUIWindow theGUI = new GUIWindow();
      theGUI.setTitle("F to C Converter");
      theGUI.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      theGUI.pack();
      theGUI.setVisible(true); //Make the window visible
   }
}
```

The main thing to note about this code is that we instantiate a `GUIWindow` instead of a `JFrame` for the main window. We then send to the `GUIWindow` the usual `JFrame` messages to set its attributes. The reason we can do this is that class `GUIWindow` extends `JFrame` (just like `ColorPanel` extends `JPanel` as discussed in previous chapters). Therefore, a `GUIWindow` is just a `JFrame` with some extra behavior.

`GUIWindow` is the main view class. It has the following responsibilities:

1. Instantiate and maintain a reference to the data model, a `Thermometer`.
2. Instantiate and maintain references to the data fields and the command button.
3. Add these widgets to the window's container, under the influence of the appropriate layout.
4. Instantiate and attach a `FarhenheitListener` to the command button.

The class `FahrenheitListener` is a private class defined within `GUIWindow`. This follows the practice established in earlier chapters, in which the code for a listener has access to the data variables of the enclosing view class. Our listener in this case must take input from one data field, use it to reset the thermometer's Fahrenheit value, and reset the other data field with the thermometer's Celsius value. Here is the code for the two new classes, heavily commented to mark the model, view, and controller:

```
/* Example 8.5 GUIWindow.java
The main view for a GUI-based temperature conversion
program that coverts from Fahrenheit to Celsius.
*/
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class GUIWindow extends JFrame{

   // >>>>>>> The model <<<<<<<<

   // Declare and instantiate the thermometer
   private Thermometer thermo = new Thermometer();

   // >>>>>>> The view <<<<<<<<

   // Declare and instantiate the widgets.
   private JLabel fahrLabel        = new JLabel("Degrees Fahrenheit");
   private JLabel celsiusLabel     = new JLabel("Degrees Celsius");
   private JTextField fahrField    = new JTextField("32.0");
   private JTextField celsiusField = new JTextField("0.0");
   private JButton fahrButton      = new JButton("Convert >>>");

   // Constructor
   public GUIWindow(){
      // Set up panels to organize widgets and
      // add them to the window
      JPanel dataPanel = new JPanel(new GridLayout(2, 2, 12, 6));
      dataPanel.add(fahrLabel);
      dataPanel.add(celsiusLabel);
      dataPanel.add(fahrField);
      dataPanel.add(celsiusField);
      JPanel buttonPanel = new JPanel();
      buttonPanel.add(fahrButton);
      Container container = getContentPane();
      container.add(dataPanel, BorderLayout.CENTER);
      container.add(buttonPanel, BorderLayout.SOUTH);
      // Attach a listener to the convert button
      fahrButton.addActionListener(new FahrenheitListener());
   }

   // >>>>>>> The controller <<<<<<<<

   private class FahrenheitListener implements ActionListener{
      public void actionPerformed(ActionEvent e){
         String input = fahrField.getText();       // Obtain input
         double fahr = Double.parseDouble(input);  // Convert to a double
         thermo.setFahrenheit(fahr);               // Reset thermometer
         double celsius = thermo.getCelsius();     // Obtain Celsius
         celsiusField.setText("" + celsius);       // Output result
      }
   }
}
```

The creation of the model and the widgets needs no further comment (although you should browse the Java API for `JButton`, `JTextField`, and `JLabel` if you are curious about setting their attributes, such as the color, text font, and so forth). However, the code in the constructor and the listener call for more detailed explanation.

Although this is a fairly simple GUI with only five widgets, we have to take some care in planning their layout. We decide that a reasonable layout would place the two labels and the two data fields in parallel rows. The single command button would then occupy the third row by itself. Recall that the default layout of a `JFrame` is a `BorderLayout`, but its five regions do not reflect the desired positions of the five widgets. Alternatively, a 2-by-2 `GridLayout` would nicely organize the rows of labels and fields, but what about the button? If we could just get all the data widgets with their grid layout into the center region of the border layout, we could put the button into its south region. The solution is to create a separate panel with a grid layout, add the data widgets to this panel, and then add the panel to the frame's center pane. We also create a new panel for the button, because adding it directly to a border layout's region would stretch the button to fill that region. This panel's flow layout causes the button to be centered with its preferred size in the panel. Finally, note that the grid layout is created with two extra parameters, which specify the amounts of horizontal and vertical padding around the widgets in pixels.

The code for the `FahrenheitListener` class represents this application's controller. When the user clicks the **Convert >>>** button, the listener's `actionPerformed` method is triggered. The first line of code uses the method `getText()` to fetch the string currently in the input field. The method `Double.parseDouble(aString)` then converts this string to a number of type `double`. The number is fed to the thermometer as a Fahrenheit value, and the equivalent Celsius value is then extracted, as in the previous examples. Finally, the method `setText(aString)` is used to output the result to the output field. Note that the number must be converted back to a string before output. Table 8-4 lists some commonly used `JTextField` methods.

TABLE 8-4
Some `JTextField` methods

JTEXTFIELD METHOD	WHAT IT DOES
`String getText()`	Returns the string currently occupying the field
`void setEditable(boolean b)`	The field is read-only by the user if `b` is `false`. The user can edit the field if `b` is `true`. The default is `true`.
`void setText(String s)`	Displays the string `s` in the field

Our first GUI program is simple as far as GUI programs go, but there was quite a lot to learn! However, the basic pattern of model/view/controller helps to organize our code and will be used in most of the GUI examples that remain in this book.

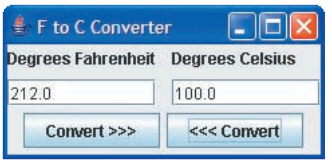
On the other hand, we now have a user interface that's both prettier and easier to use than the corresponding query-driven interface. And we didn't even have to write a loop like we did in the terminal-based version of Section 8.1. This illustrates an important point about GUI-based programs: unlike programs based on terminal I/O or even dialog box I/O, real GUI programs are ***event-driven***. This means that when a GUI program opens, it just sits there and waits for events (the user clicking the mouse, typing characters into a field, etc.). You can imagine the JVM running a loop behind the scenes, which halts only when the user clicks the window's close icon. When any other event occurs within this loop, control shifts to the GUI program's method for handling that event, if there is one. When that method finishes, control returns to the JVM's loop. If there are no methods for handling events, they are ignored.

Making Temperature Conversion Go Both Ways

The menu-driven version of the temperature conversion program in Section 8.2 allows the user to convert a Fahrenheit value to a Celsius value or a Celsius value to a Fahrenheit value. This change involved a major restructuring of the query-driven program of Section 8.1, as well as a major change in the user interface. Let's examine the changes in the GUI version to make the conversion go both ways.

The only change to the user interface is the addition of a second button, as shown in Figure 8-5. When the user clicks the **Convert >>>** button, the program uses the Fahrenheit field for input and the Celsius field for output, as before. When the user clicks the new button, **<<< Convert**, the program uses the Celsius field for input and the Fahrenheit field for output.

FIGURE 8-5
A temperature converter that goes both ways



There are only minor changes to the code as well. The `GUIWindow` class

- Declares and instantiates the second button:

```
private JButton celsiusButton  = new JButton("<<< Convert");
```

- Adds this button to the button panel:

```
buttonPanel.add(celsiusButton);
```

- Creates a listener object and attaches it to the button:

```
celsiusButton.addActionListener(new CelsiusListener());
```

- Defines a separate listener class that converts from Celsius to Fahrenheit:

```
// Example 8.6: Listener to convert Celsius to Fahrenheit
private class CelsiusListener implements ActionListener{
   public void actionPerformed(ActionEvent e){
      String input = celsiusField.getText();          // Obtain input
      double celsius = Double.parseDouble(input);     // Convert to a double
      thermo.setCelsius(celsius);                     // Reset thermometer
      double fahr = thermo.getFahrenheit();           // Obtain Fahrenheit
      fahrField.setText("" + fahr);                   // Output result
   }
}
```

In short, we simply add a new command button and the corresponding controller to take action when the user clicks this button. No matter how many new widgets are added to the GUI, if we maintain the correspondence between a widget and its controller, the basic structure of the program will never change. And still no loop!

Making Temperature Conversion Robust

Section 8.5 discussed a strategy for recovering from format errors in numeric input. A similar technique can be applied to catch input errors with text fields in GUI programs. The methods `Double.parseDouble(aString)` and `Integer.parseInt(aString)` are used to convert strings to numbers of type `double` and `int`, respectively. If either of these methods encounters a number format error, it throws an exception. Let's modify the code that converts from Fahrenheit to Celsius to catch a number format exception and recover from it. The code for `actionPerformed` is now enclosed in a `try-catch` statement. If an exception is thrown, control is shifted to the `catch` clause, which pops up a message dialog box to inform the user. Here is the code for the modified listener:

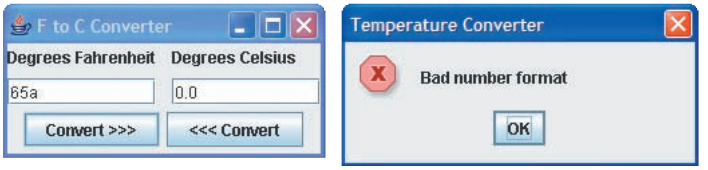
```
// Example 8.7: A robust listener for number format errors
private class FahrenheitListener implements ActionListener{
   public void actionPerformed(ActionEvent e){
      try{
         String input = fahrField.getText();        // Obtain input
         double fahr = Double.parseDouble(input);   // Convert to a double
         thermo.setFahrenheit(fahr);                // Reset thermometer
         double celsius = thermo.getCelsius();      // Obtain Celsius
         celsiusField.setText("" + celsius);        // Output result
      }catch(Exception ex){
         JOptionPane.showMessageDialog(GUIWindow.this,
                                       "Bad number format",
                                       "Temperature Converter",
                                       JOptionPane.ERROR_MESSAGE);
      }
   }
}
```

The code for the normal processing of this command is nested in the `try` clause. If an exception is thrown in the call of `parseDouble`, control is transferred immediately to the `catch` clause, which pops up an error message dialog box.

Note that the first parameter of the method `showMessageDialog` is the expression `GUIWindow.this`, which in this context refers to the current instance of the `GUIWindow` class. This parameter allows the JVM to track the "parent" of the dialog box, that is, the main window from which it was opened. In the examples of earlier chapters, the first parameter had been `null` because no parent window was involved. If we had simply used the parameter `this` instead, the compiler would think that we meant the current instance of `FahrenheitListener`, which is not a GUI component.

Finally, `showMessageDialog` includes two other parameters for the dialog box's title and its type, which displays an error icon, as shown in Figure 8-6.

FIGURE 8-6
Responding to a number format error



(a) Main window (b) Message box

EXERCISE 8.6

1. Write a code segment that obtains an input from a text field called `inputField`, converts the input to an integer, and outputs the square root of the integer to a text field called `outputField`.
2. Assume that an action listener has been added to a button. Which method runs when the user clicks the button?
3. Describe the roles and responsibilities of the model, view, and controller classes in a GUI program.

SUMMARY

In this chapter, you learned:

- A terminal input/output (I/O) interface can be extended to handle repeated sets of inputs, by using either a query-based pattern or a menu-driven pattern.
- A graphical user interface (GUI) allows the user to interact with a program by displaying window objects and handling mouse events.
- In a terminal-based program, the program controls most of the interaction with the user, whereas GUI-based programs are driven by user events.
- The two primary tasks of a GUI-based program are to arrange the window objects in a window and handle interactions with the user.

VOCABULARY *Review*

Define the following terms:

Application	Event-driven	Menu-driven progam
Controller pattern	Format flag	Model view
Data model	Format specifier	Query-controlled input

REVIEW *Questions*

FILL IN THE BLANK

Complete each of the following statements by writing your answer in the blank provided.

1. In contrast to terminal I/O programs, GUI programs are _____ driven.
2. A button allows the user to select a(n) _____.
3. Two types of window objects that support numeric I/O are a(n) _____ and a(n) _____.

4. A window object that supports the I/O of a single line of text is a(n) _____.
5. A separate window that pops up with information is a(n) _____.

PROJECTS

PROJECT 8-1

Newton's method for computing the square root of a number consists of approximating the actual square root by means of a set of transformations. Each transformation starts with a guess at the square root. A better approximation is then (guess + number / guess) / 2. This result becomes the guess for the next approximation. The initial guess is 1. Write a query-driven program that allows the user to enter a number and the number of approximations to compute its square root.

PROJECT 8-2

Modify the program of Project 8-1 so that the user can view the successive approximations. (*Hint*: Build a formatted string of the approximations during the computation.)

PROJECT 8-3

John has $500 to invest. Sue knows of a mutual fund plan that pays 10 percent interest, compounded quarterly (that is, every 3 months, the principal is multiplied by the 2.5 percent and the result is added to the principal; more generally, the amount of gain each quarter is equal to current balance * (1 + interest rate / 400)). Write a program that tells John how much money will be in the fund after 20 years. Make the program general; that is, it should take as inputs the interest rate, the initial principal, and the number of years to stay in the fund. The output should be a table whose columns are the year number, the principal at the beginning of the year, the interest earned, and the principal at the end of the year.

PROJECT 8-4

The TidBit Computer Store has a credit plan for computer purchases. There is a 10 percent down payment and an annual interest rate of 12 percent. Monthly payments are 5 percent of the listed purchase price minus the down payment. Write a program that takes the purchase price as input. The program should display a table of the payment schedule for the lifetime of the loan. Use appropriate headers. Each row of the table should contain the following items:

- Month number (beginning with 1)
- Current total balance owed
- Interest owed for that month
- Amount of principal owed for that month
- Payment for that month
- Balance remaining after payment

The amount of interest for a month is equal to balance * rate / 12. The amount of principal for a month is equal to the monthly payment minus the interest owed.

PROJECT 8-5

Modify the final temperature conversion program of Section 8.6 so that it displays the results of each conversion rounded to the nearest hundredth of a degree.

PROJECT 8-6

Write a GUI program that takes a radius as input. The outputs, displayed in separate fields, are the area of a circle, the surface area of a sphere, and the volume of a sphere of this radius.

CRITICAL *Thinking*

A company approaches you about the need for a program and wonders whether to ask for a terminal-based user interface or a graphical user interface. Discuss the issues involved in choosing between these two interfaces from a client's perspective.