Chapter 3

Probing Questions

In the preceding chapter, we learned how to give commands in Java. Using method invocations, you can now write programs that create GUI components and then command them to change in various ways. In this chapter, we show that method invocations can also be used to ask questions. For example, at some point in a program you might need to determine what a user has typed into a text field. We will learn that you can ask a text field to tell you what it currently contains by invoking a method named `getText`. A similar method can be used to ask a menu which of its items is currently selected. Such methods are called *accessor methods*. They are used to access information about the state of an object. In contrast, all the methods we introduced in the last chapter are used to change object states. Such methods are called *mutator methods*.

There is one important kind of question that cannot be asked using an accessor method. Imagine a program that displays two buttons in its interface. One button might be labeled "Print" while the other is labeled "Quit". If a user clicks one of these buttons, the code in the program's `buttonClicked` method will be executed. Somehow, this method will have to ask which button was clicked. We will see that rather than depending on method invocations to determine this information, Java instead provides a way to directly associate a name with the GUI component that causes an event-handling method to be invoked.

Finally, after discussing the basics of using both mutator and accessor methods, we will explore the collection of methods that can be used to manipulate GUI components. In addition, we will introduce one additional type of GUI component, the `JTextArea` which is used to display multiple lines of text.

3.1 Accessor Methods

To introduce the use of accessor methods, we will construct a program that implements aspects of a feature found in many web browsers, the bookmarks or favorites menu. Obviously, we are not going to try to present the code for a complete web browser in this chapter. Instead, we will focus on exploring the type of code that might be included in a web browser to allow a user to add the addresses of interesting web sites to a menu.

The interface for the program we want to examine will be quite simple. There will be a `JTextField` in which a user can type items that should be added to the favorites menus. We will assume that the user types addresses of web sites into this text field, but our program will actually work the same way no matter what form of text the user chooses to enter. There will also be a

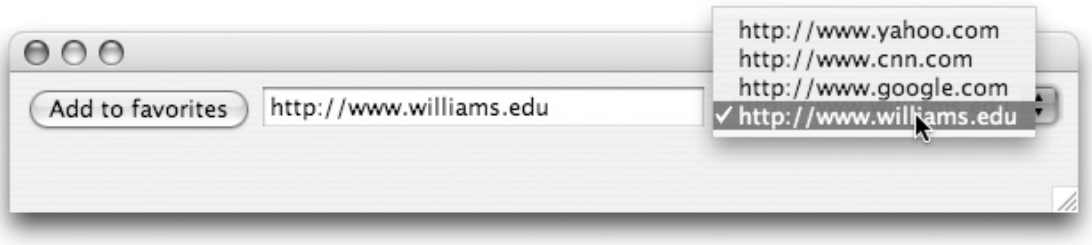

Figure 3.1: Interface for program implementing a "favorites" menu

`JComboBox` that will display the list of items the user has identified as "favorite" web site addresses. Finally, there will be a button labeled "Add". When this button is pressed, the address currently in the text field will be added to the menu as a new item. An image of what the program's interface might look like while the user is actually selecting an item from the favorites menu is shown in Figure 3.1. Of course, selecting an item from the menu will have no real effect. The only reason we show the program in this state is so that you can see that items have been added to the menu, including the latest item, `www.williams.edu`, which still appears in the `JTextField`.

The only detail of implementing this program that will be new is the use of the accessor method named `getText` to determine what the user has typed into the text field. In some respects even this will seem fairly familiar, because using an accessor method is very much like using a mutator method. To invoke either a mutator method or an accessor method we must write a phrase of the form

name.action(. . . *arguments* . . .)

Accordingly, before we can use `getText` to find out what is in a `JTextField` we will have to associate a name with the `JTextField`, much as we had to associate names with buttons and labels in the last chapter in order to use mutator methods.

Assume that we include the following declarations in our program:

```
// A place where the user can type in a web site address
private JTextField address;

// Size of field used to enter addresses
private final int ADDR_LENGTH = 20;

// Menu that will hold all the addresses identified as favorite sites
private JComboBox favorites;
```

and that we include code in our constructor to create a text field and a menu and to associate them with these names:

```
address = new JTextField( ADDR_LENGTH );
favorites = new JComboBox();
```

Finally, assume that we include code in the constructor to add these two components and a `JButton` to the `contentPane` Note that in the assignment statement for `favorites` we use a `JComboBox`

construction that includes no arguments. This type of construction produces a menu that initially contains no items. The only detail we have left to consider is what code to put in the `buttonClicked` method to actually add the contents of the text field as a new item in the menu.

There is a mutator method named `addItem` that is used to add entries to a menu. If the invocation

```
favorites.addItem( "http://www.weather.com" );
```

is executed, the computer will add the address `http://www.weather.com` as a new item in the menu. Of course, while we will want to use this method, we cannot use it in this way. We cannot just type in the item we want to add in quotes as an argument because we won't know what needs to be added until the program is running. We have to provide something in the invocation of the `addItem` method that will tell the computer to ask the text field what it currently contains and use that as the argument to `addItem`.

Java lets us do this by using an invocation of the `getText` accessor method as an argument to the `addItem` method. The complete command we need to place in the `buttonClicked` method will therefore look like

```
favorites.addItem( address.getText() );
```

The complete program is shown in Figure 3.2.

3.2 Statements and Expressions

It is important to remember that accessor methods like `getText` serve a very different function than mutator methods. A mutator method instructs an object to change in some way. An accessor method requests some information about the object's current state. As a result, we will use accessor methods in very different contexts than mutator methods. With this in mind, it is a good time for us to take a close look at some aspects of the grammatical structure of Java programs and clearly distinguish between two types of phrases that are critical to understanding this structure.

The body of a method definition consists of a sequence of commands. These commands are called *statements* or *instructions*. We have seen two types of phrases that are classified as statements so far: assignment statements such as

```
message = new JLabel( "Click the button above" );
```

and method invocations such as

```
contentPane.add( new JButton( "Click Here") );
```

Simply asking an object for information is rarely a meaningful command by itself. We would never write a statement of the form:

```
address.getText();
```

One might mistakenly think of such a method invocation as a command because it does tell the computer to do something — to ask the `JTextField` for its contents. It would be a useless command, however, because it does not tell the computer what to do with the information requested. For this reason, invocations that involve mutator methods are used as statements, but invocations that involve accessor methods are not.

```
import squint.*;
import javax.swing.*;

/*
 * A program to demonstrate the techniques that would be used
 * to enable a user to add addresses to a favorites menu in a
 * web browser.
 */
public class FavoritesMenu extends GUIManager {
  // Change these to adjust the size of the program's window
   private final int WINDOW_WIDTH = 600, WINDOW_HEIGHT = 100;

   // A place where the user can type in a web site address
   private JTextField address;

   // Size of field used to enter addresses
   private final int ADDR_LENGTH = 20;

   // Menu that will hold all the addresses identified as favorite sites
   private JComboBox favorites;

   /*
    *  Create the program window and place a button, text field and
    *  a menu inside.
    */
   public FavoritesMenu() {
     this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

      address = new JTextField( ADDR_LENGTH );
      favorites = new JComboBox();

      contentPane.add( new JButton( "Add to favorites" ) );
      contentPane.add( address );
      contentPane.add( favorites );
   }

   /*
    *  Add contents of JTextField to JComboBox
    */
   public void buttonClicked( ) {
      favorites.addItem( address.getText() );
   }
}
```

Figure 3.2: Code for program to simulate a favorites menu

Invocations that involve accessor methods are instead used in contexts where Java expects us to describe an object or value. There are many contexts in which such phrases are used. We have already noted that this is the type of phrase expected to appear on the right side of an assignment statement. The function of the assignment is then to associate the name on the left side of the equal sign with the object or value described by this phrase. We also use such phrases when we describe the arguments provided in method invocations and constructions. Phrases of this type are called *expressions*.

At this point, we have seen five types of phrases that can be used as expressions:

1) accessor method invocations The use of accessor method invocations to describe values was first encountered in the preceding section. A phrase like

```
address.getText()
```

is an expression that describes a value by telling some object (the text field `address` in this case) to provide the desired information.

2) literals Expressions that directly and explicitly identify particular values are called *literals*. For example, in the statements

```
contentPane.add( new JButton( "Click here" ) );
contentPane.add( new JTextField( 20 ) );
```

the number 20 and the string `"Click here"` are examples of literals.

3) constructions Recall that a construction is a phrase of the form

```
new Type-of-object( ... arguments ... )
```

that tells Java to create a new object. In addition to creating an object, a construction serves as a description of the object that is created. As such, constructions are expressions. In the two examples included in our discussion of literals, the constructions of a `JButton` and a `JTextField` are used to describe the arguments to the content pane `add` method.

4) variable names Almost as soon as we introduced instance variable names, we showed that we could replace a command like

```
contentPane.add( new JTextField( 20 ) );
```

with the pair of commands

```
message = new JTextField( 20 );
contentPane.add( message );
```

In the second of these two lines, the variable name `message` has taken on the role of describing the argument to the `add` method. Thus, variable names can also be used as expressions.

5) formulas In the `buttonClicked` method of the `TouchCounter` program shown in Figure 2.10, we saw the statement

```
numberOfClicks = numberOfClicks + 1;
```

In this instruction, we describe a value by telling Java how to compute it using the addition operation. The phrase

```
numberOfClicks + 1
```

is an example of an expression that is formed by combining simpler expressions (the variable `numberOfClicks` and the literal `1`) using the addition operator. An expression like this is called a *formula*. We will see that Java supports many operators, including operators for the usual arithmetic operations addition (`+`), subtraction (`-`), division (`/`), and multiplication (`*`).

Java allows us to use whatever form of expression we want to describe a value on the right hand side of an assignment statement or as an argument in an invocation or a construction. It doesn't care whether we use a literal, variable, invocation, construction or a formula. It is very picky, however, about the type of information the expression we use describes. For example, the parameter provided to the content pane's add method must be a GUI component. As a result, we can say

```
contentPane.add( new JLabel( "Click here" ) );
```

but we cannot say

```
contentPane.add( "Click here" );
```

The literal `"Click here"` describes a `String`. While a string can be used as a parameter in a construction that creates a GUI component, the string itself is not a GUI component and therefore the literal `"Click here"` will not be accepted as a valid argument for the `add` method.

3.3 Working with GUI Components

In the preceding sections, we have used examples involving GUI components to introduce the ways in which method invocations can be used within Java programs. Now that we have explored the basics of using method invocations both as statements and expressions, we want to explore the collection of methods that can be used to manipulate GUI components more thoroughly. In addition, we will introduce a new type of GUI component, the `JTextArea` which is similar to the `JTextField` but allows us to display multiple lines of text.

The complete collection of methods associated with the GUI components we have been using is quite extensive. We will not cover all of these methods. We will limit our attention to a relatively small selection of methods that provides access to the essential functions of these GUI components. The curious reader can consult the online documentation for the GUI components provided by the Swing library at

```
http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/package-summary.html
```

3.3.1 Common Features

There are some methods that can be applied to any GUI component. We have already introduced several such methods. When we first introduced `setEnabled`, we mentioned that this method could be applied to any GUI component. `setEnabled` can be used to enable or disable any type of GUI component. In our examples, we only used the `setForeground` method to set the color of text displayed in a `JButton`. In fact, this method and its close relative, `setBackground`, can be applied to any GUI component. Thus, we can use these methods to change the colors used to display buttons, labels and menus.

In addition to these mutator methods that are shared by all types of GUI components, there are also accessor methods that can be applied to components of any type. One example is a counterpart to `setEnabled`. It is an accessor method named `isEnabled`. We saw that `setEnabled` expects us to specify either `true` or `false` as its argument depending on whether we want the component involved to be enabled or disabled. If `oneButton` and `anotherButton` are variable names associated with `JButton`s, then

```
oneButton.isEnabled()
```

is an expression that either describes the value `true` or `false` depending on whether `oneButton` is currently enabled or disabled. Therefore, the statement

```
anotherButton.setEnabled( oneButton.isEnabled() );
```

could be used to make sure that `anotherButton`'s state is the same as `oneButton`'s state as far as being enabled or disabled.

3.3.2 Menu Methods

We have already introduced one method used withthe `JComboBox` class, the `addItem` method that inserts a new entry into a menu. In addition, the `JComboBox` class provides an extensive collection of methods for removing items, determining which items are currently displayed in a menu, and changing the item that is selected in a menu. We will present several of these methods in this section. We will start with what we consider the most important method in the collection, `getSelectedItem`, a method used to determine which menu item is currently selected.

Using the `getSelectedItem` Method

`getSelectedItem` is an accessor method. To illustrate its use, we will add a bit of functionality to the `FavoritesMenu` program we presented in Figure 3.2. Our earlier version of this program allowed us to add items to a menu, but it didn't react in any significant way if we selected an item from the menu. If the menu were part of an actual web browser, we would expect it to load the associated web page when we selected an item from the favorites menu. We will set a simpler goal. We will modify the program so that when a menu item is selected, the web address associated with the menu item is placed in the `JTextField` included in the program's display.

A program can tell a text field to change its contents by applying the `setText` method to the text field. For example, executing the invocation

```
address.setText( "http://www.aol.com" );
```

would place the address of the web site for America Online in the `JTextField`.

This use of `setText` should seem familiar. In an earlier example, we applied `setText` in a similar way to change the words displayed by a `JLabel`. In fact, we can use `setText` to change the text displayed by either a `JLabel`, a `JTextField`, or a `JButton`.

Recall that whenever an item is selected from a menu, any instruction placed in the method named `menuItemSelected` will be executed. This suggests that we can make our program display the contents of an item when it is selected by adding a method definition of the form

```
public void menuItemSelected( ) {
   address.setText( ... );
}
```

to the `FavoritesMenu` program. All we have to do is figure out what argument to provide where we typed "..." above. Given that we just explained that the `getSelectedItem` method can be used to determine which item is selected, an invocation of the form

```
favorites.getSelectedItem()
```

might seem to be an obvious candidate to specify the parameter to `setText`. Unfortunately, this will NOT work. Your IDE will reject your program as erroneous if you use this parameter. The problem arises because `setText` expects a `String` as its argument but `getSelectedItem` doesn't always return a `String`.

You can probably imagine that it is sometimes useful to create a menu whose items are more complex than strings — a menu of images for example. The `JComboBox` provides the flexibility to construct such menus (though we have not exploited it yet). When working with such menus, the result produced by invoking `getSelectedItem` will not be a `String`. Java's rules for determining when an expression is a valid parameter are quite conservative in accounting for this possibility. Even though it is obvious that the result of using `getSelectedItem` in this particular program has to be a `String`, Java will not allow us to use an invocation of `getSelectedItem` to specify the parameter to `setText` because such an invocation will not always produce a `String` as its result in every program.

There is another accessor method that we can use to get around this problem. It is named `toString` and it can be applied to any object in a Java program. It tells the object to provide a `String` that describes itself. The descriptions provided by invoking `toString` can be quite cryptic in some cases. If the object that `toString` is applied to is actually a `String`, however, then the result produced by the invocation is just the `String` itself. So, by applying the `toString` method to the result produced by applying `getSelectedItem` we can provide the argument we want to the `setText` method.

There is one last subtlety remaining. Up until this point we have stated that the form of an invocation in Java was

name.action(. . . *arguments* . . .)

If this were true, then we would first have to associate a name with the result produced by the invocation of `getSelectedItem` before we could apply the `toString` method to that result. Fortunately, we haven't told you the whole truth about this.

The actual description for the form of an invocation in Java is

expression.action(. . . *arguments* . . .)

That is, just as you can use any of the five forms of expression — names, literals, constructions, invocations, and formulas — to specify the arguments in an invocation, you can use any form of expression to describe the object to which the action should be applied. So, to apply the `toString` method to the result of applying the `getSelectedItem` method we can write

```
favorites.getSelectedItem().toString()
```

This is an expression that is acceptable as a specification for the argument to `setText`. As a result, we can extend the program as we desire by adding the following method declaration:

```
public void menuItemSelected( ) {
   address.setText( favorites.getSelectedItem().toString() );
}
```

Examining Menu Items

There are two methods that can be applied to a `JComboBox` to find out what items are currently in the menu. While these methods are not very important in programs that display menus that contain fixed sets of items, they can be helpful in programs where the user can add and delete menu items.

The first of these methods is named `getItemCount`. It is an accessor method that returns the number of items currently in a menu. The value returned will always be an `int`.

The other method is named `getItemAt`. It returns the item found at a specific position in the menu. The items in a `JComboBox` are numbered starting at 0. Thus, the invocation

```
favorites.getItemAt( 0 )
```

would return the first item in the menu and

```
favorites.getItemAt( 3 )
```

would produce what one might normally consider the fourth item in the menu. Because one can create menus that contain items that are not `String`s, Java will not allow you to use an invocation of `getItemAt` in a context where a `String` is required. As we explained in the preceding section, if you need to use `getItemAt` in such a context, you can do so by combining its invocation with an invocation of `toString`.

There is also a method named `getSelectedIndex` which returns the position of the currently selected item counting from position 0.

Changing the Selected Menu Item

Java provides two methods to change the item that is selected within a menu. The first, `setSelectedIndex`, expects you to specify the position of the item you would like to select numerically. Again, items are numbered starting at 0 so

```
favorites.setSelectedIndex( 0 );
```

would select the first item in the `favorites` menu while

```
favorites.setSelectedIndex( favorites.getItemCount() - 1 );
```

would select the last item in the menu.

The `setSelectedItem` method allows you to select a new item by providing its contents as an argument. For example, If a menu named `indecision` contained the items "Yes", "No", and "Maybe so" like the menus shown in Figure 1.12, then the statement

```
indecision.setSelectedItem( "No" );
```

could be used to select the middle item.

Removing Items from a Menu

The simplest way to remove items from a menu is to remove all of them. This can be done using a method named `removeAllItems` as in the statement

```
favorites.removeAllItems();
```

It is also possible to remove one item at a time either by specifying the position of the item to be removed or its contents. When removing by position, the method to use is `removeItemAt`. For example, the invocation

```
favorites.removeItemAt( 0 );
```

would remove the first item from the menu. The method `removeItem` can be used to remove an item based on its contents as in

```
indecision.removeItem( "Maybe so" );
```

3.3.3 Components that Display Text

We have already encountered two types of GUI components that can be used to display text, `JTextField`s and `JLabel`s. `JTextField`s and `JLabel`s are obviously different in many ways. A program's user can type new text into a text field, while the information displayed in a `JLabel` can only be changed by invoking the `setText` method from within the program. At the same time, however, these two types of GUI components share many similarities. For example, we have seen that the `setText` method can be applied to either a `JLabel` or a `JTextField`. Similarly, the `getText` accessor method can be applied to either a `JTextField` or a `JLabel` to access its current contents.

`JTextArea`s and `JScrollPane`s

There is a third type of GUI component used to display text that shares even more methods with `JTextField`. This type of component is called a `JTextArea`. A `JTextArea` provides a convenient way to display multiple lines of text in a program's window. When we construct a `JTextField`, we include a number in the construction that indicates how many characters wide the field should be. When we construct a `JTextArea`, we specify how many lines tall the field should be in addition to specifying how many characters wide. The construction involves two arguments with the first specifying the number of lines. Thus, a construction of the form

```
new JTextArea( 20, 40 )
```

```
/*
 * Class HardlyAWord - A program that presents a text area in
 * which a user can enter and edit text.
 *
 */
public class HardlyAWord extends GUIManager {

   // Change these values to adjust the size of the program's window
   private final int WINDOW_WIDTH = 300, WINDOW_HEIGHT = 150;

  /*
   * Place a JTextArea in the window
   */
   public HardlyAWord() {
      // Create window to hold all the components
      this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

      contentPane.add( new JTextArea( 6, 20 ) );
   }
}
```

Figure 3.3: The world's smallest word processing program

would create a text area tall enough for 20 lines and wide enough for 40 characters in any single line.

A program that uses a `JTextArea` is shown in Figure 3.3. It is a very simple program. In fact it is about the simplest program you could imagine writing that actually does something that is at least somewhat useful. When the program runs it displays a window containing an empty text area as shown in Figure 3.4.

While the empty text area in this window may not look very impressive, it actually provides a fair amount of functionality. You can click the mouse in the text area and start typing. It won't automatically start a new line for you if you type in too much to fit within its width, but you can start new lines by pressing the return key. If you make a mistake, you can reposition the cursor by clicking with the mouse. You can even cut and paste text within the window. Figure 3.5 shows how the program window might look after we had used these features to enter the opening words of an exciting little story.

There is, however, a difficulty. If we keep typing but press the return key less frequently so that the complete text entered would look like

```
Once upon a time, in a small text
area sitting on a computer's screen
a funny thing happened.  As the
computer's user typed and typed
and typed, the small text area got bigger and bigger
until it no longer fit very well in a little window.
```

Figure 3.4: The initial state of the `HardlyAWord` program in execution

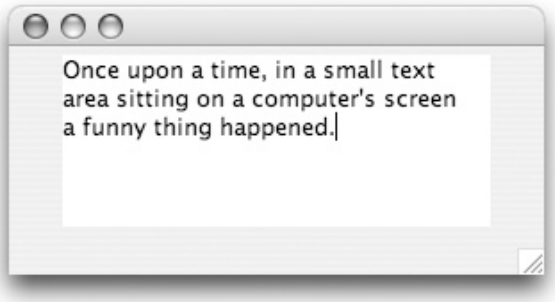

Figure 3.5: The state of `HardlyAWord` as text is entered

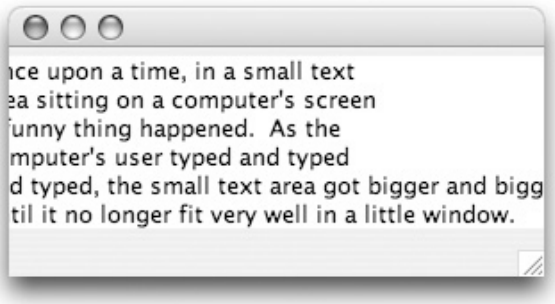

Figure 3.6: The result of entering too much text

the contents of the program's window will come to look like the image shown in Figure 3.6.

The problem is that the numbers included in a `JTextArea` construction to specify how many lines high and how many characters wide it should be are treated as specifications of its minimum size. As long as the text placed within the area fits within these minimum values, the text area's size will remain fixed. If the text exceeds the minimum values, however, the text area will automatically grow to be big enough to fit the text. If necessary, it will grow so big that it might no longer fit within the program's window, as we can see in Figure 3.6.

As a result of this problem, it is unusual to use a `JTextArea` by itself. Instead, it is typical to use a `JTextArea` together with another type of component called a `JScrollPane`. A `JScrollPane` can hold other components just as a `JPanel` can. If the components it holds become too big to fit in the space available, the `JScrollPane` solves the problem by providing scroll bars that can be used to view whatever portion of the complete contents of the scroll pane the user wants to see.

It is very easy to associate a `JScrollPane` with a `JTextArea`. We simply provide the `JTextArea` as an argument in the construction of the `JScrollPane`. For example, to use a scroll pane in our `HardlyAWord` program we would just replace the line

```
contentPane.add( new JTextArea( 6, 20 ) );
```

with the line

```
contentPane.add( new JScrollPane( new JTextArea( 6, 20 ) ) );
```

When this new program is run, its window will initially look the same as it did before (see Figure 3.4). A `JScrollPane` does not add scroll bars until they become necessary. Once we entered the text we showed earlier, however, the window would take on the appearance shown in Figure 3.7. We still can't see all the text that has been entered, but we can use the scroll bars to examine whatever portion is of interest.

`JTextArea` Methods

In some programs, `JTextArea`s are used to provide a place in which the program's user can enter text as in our `HardlyAWord` example. In such programs, the most important method associated with `JTextArea`s is the accessor method `getText`. Just as it does when applied to a `JTextField`, this method returns a `string` containing all of the text displayed within the text area.

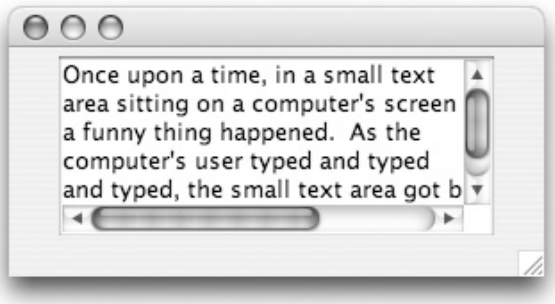

Figure 3.7: A `JTextArea` displayed within a `JScrollPane`

In other programs, a `JTextArea` may be used to display information to the user. If all the information to be displayed is available at once, the `setText` method can be used in the same manner it might be used with a `JTextField` or `JLabel`. Frequently, however, the information to be displayed is added to the `JTextArea` bit-by-bit as it becomes available. In this case, another method named `append` is more useful than `setText`. While `setText` replaces whatever information had been displayed in a text area by the `string` provided as its argument, `append` simply adds the text provided as its argument to the information that was already displayed in the text area.

Another method that is often used by programs that use `JTextArea`s to present information to users is named `setEditable`. By default, a user can edit the text in a `JTextArea` using the mouse and keyboard as we explained in our discussion of the `HardlyAWord` program. If a `JTextArea` is being used strictly as a means of communicating information to a user, it may be best to prevent the user from editing its contents. Otherwise, the user might accidentally delete or change information they might like to see later. If `displayArea` is a variable associated with a `JTextArea`, then the statement

```
displayArea.setEditable( false );
```

will tell the computer not to let the user edit the contents of the text area. If appropriate, at some later point the invocation

```
displayArea.setEditable( true );
```

can be used to enable editing as usual. The `setEditable` method can also be a applied to a `JTextField`.

To illustrate the use of `append` and `setEditable`, suppose that a program that included a menu in its interface wanted to display the history of items that had been selected from this menu. To keep things simple, we will write the example code using a familiar example of a menu, our "Yes/No/Maybe so" menu. An image of how the program's interface might look after its user had selected items from the menu at random 10 times is shown in Figure 3.8.

The complete code for the program is shown in Figure 3.9. The `JTextArea` in which the selected menu items are displayed is named `log`. The `setEditable` method is applied in the constructor just before the `log` is added to the content pane so that it will never be editable while it is visible on the screen.

The `append` method is used in `menuItemSelected`. There are two interesting features of the argument passed to `append`. First, since we are using `getSelectedItem`, we must use the `toString`

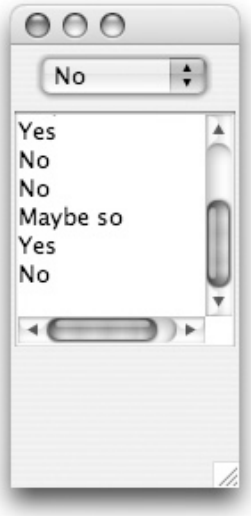

Figure 3.8: Sample of a menu item selection log

method to assure Java that we will be working with a `String` even if `getSelectedItem` returns some other type of object. Second, rather than just using `append` to add the menu item returned by `getSelectedItem` to the `log`, we instead first use the "+" operator to concatenate the text of the menu item with the strange looking `String` literal `"\n"`.

To appreciate the purpose of the literal `"\n"`, recall that when we first introduced the concatenation operation in the example code

```
"I'm touched " + numberOfClicks + " time(s)"
```

we stressed that if we did not explicitly include extra spaces in the literals `"I'm touched "` and `" time(s)"`, Java would not include any spaces between the words "touched" and "time(s)" and the numerical value of `numberOfClicks`. Similarly, if we were to simply tell the `append` method to add the text of each menu item selected to the `JTextArea` by placing the command

```
log.append( indecision.getSelectedItem().toString() );
```

in the `menuItemSelected` method, Java would not insert blanks or start new lines between the items. The text of the menu items would all appear stuck together on the same line as shown in Figure 3.10.

To have the items displayed on separate lines, we must add something to each item we append that will tell the `JTextArea` to start a new line. The literal `"\n"` serves this purpose. Backslashes are used in `String` literals to tell Java to include symbols that can't simply be typed in as part of the literals for various reasons. The symbol after each slash is interpreted as a code for what is to be included. The combination of the backslash and the following symbol is called an *escape sequence*. The "`n`" in the escape sequence `"\n"` stands for "new line". There are several other important escape sequences. `\"` can be included in a `String` literal to indicate that the quotation mark should be included in the `String` rather than terminating the literal. For example, if you wanted to display the two lines

```
/*
 *  Class MenuLog - Displays a history of the items a user selects
 *  from a menu
 */
public class MenuLog extends GUIManager {

   private final int WINDOW_WIDTH = 100, WINDOW_HEIGHT = 270;

   // Dimensions of the display area
   private final int DISPLAY_WIDTH = 10, DISPLAY_HEIGHT = 8;

   // Used to display the history of selections to the user
   private JTextArea log;

   // Menu from which selections are made
   private JComboBox indecision;

  /*
   *  Place the menu and text area in the window
   */
   public MenuLog() {
     this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

      indecision =  new JComboBox( new String[]{ "Yes", "No", "Maybe so"} );
      contentPane.add( indecision );

      log = new JTextArea( DISPLAY_HEIGHT, DISPLAY_WIDTH );
      log.setEditable( false );
      contentPane.add( new JScrollPane( log ) );
   }

   /*
    *  Add an entry to the log each time a menu item is selected
    */
    public void menuItemSelected( ) {
       log.append( indecision.getSelectedItem().toString() + "\n" );
    }
}
```

Figure 3.9: A program using a text area to display a simple log

Figure 3.10: The result of appending `String`s without using `"\n"`

```
What did he say?
He said "No."
```

in the `JTextArea` `log`, you could use the command

```
log.setText( "What did he say?\nHe said \"No.\"" );
```

Finally, `\\` is used to include a backslash as part of a literal.

Using and Simulating Mouse Actions

Within a `JTextField` or `JTextArea`, the position of the text insertion point or "caret" can be changed by clicking with the mouse. It is also possible to select a segment of text by dragging the mouse from one end of the segment to the other. Java provides accessor methods that can be used to obtain information related to such user actions and mutator methods that make it possible to select text and to reposition the caret under program control.

The accessor method `getSelectedText` returns a string containing the text (if any) that the user has selected with the mouse. The method `getCaretPosition` returns a number indicating where the insertion point is currently located within the text.

A program can change the selected text using the methods `selectAll` and `select`. The first method takes no arguments and simply selects all the text currently in the text area or text field. The `select` method expects two arguments describing the starting and ending positions within the text of the segment that should be selected.

When you type text while using a program that has several text fields and/or text areas in its interface, the computer needs some way to determine where to add the characters you are typing. As a user, you can typically control this by clicking the mouse in the area where you want what you type to appear or by pressing the tab key to move from one area to another. A text field or text area selected to receive typed input in this way is said to have the *focus*. It is also possible to

determine which component should have the focus by executing a command within the program using a method named `requestFocus`.

3.3.4 Summary of Methods for GUI Components

In the preceding chapters and sections, we have introduced several types of GUI components and many methods for manipulating them. Here, we summarize the details of the methods available to you when you work with the GUI components we have introduced. In fact, this section provides a bit more than a summary. In addition to describing all the methods we have presented thus far, it includes some extra methods that are less essential but sometimes quite useful. We hope this will serve as a reference for you as you begin programming with these tools.

In the description of each method, we provide a prototype of an invocation of the method. In these prototypes we use variable names like `someJButton` and `someJComboBox` with the assumption that each of these variable names has been declared to refer to a component of the type suggested by its name and associated with an appropriate component by an assignment statement. In situations where a method can be applied to or will accept as a parameter a component of any type, we use a variable name like `someComponent` to indicate that any expression that describes a GUI component can be used in the context where the name appears.

Another convention used in these prototypes involves the use of semicolons. We will terminate prototypes that involve mutator methods with semicolons to suggest that they would be used as statements. Phrases that involve accessor methods and would therefore be used as expressions will have no terminating semicolons.

Methods Applicable to All Components

We begin by describing methods that can be applied to GUI components of any type.

```
someComponent.setForeground( someColor );
someComponent.setBackground( someColor );
```

The `setForeground` and `setBackground` methods can be used to change the colors used when drawing a GUI component on the display. The foreground color is used for text displayed. The actual parameter provided when invoking one of these methods should be a member of the class of `Color`s. In particular, you can use any of the names `Color.BLACK`, `Color.DARK_GRAY`, `Color.LIGHT_GRAY`, `Color.GRAY`, `Color.CYAN`, `Color.MAGENTA`, `Color.BLUE`, `Color.GREEN`, `Color.ORANGE`, `Color.PINK`, `Color.RED`, `Color.WHITE`, or `Color.YELLOW` to specify the arguments to invocations of these methods.

```
someComponent.setEnabled( true );
someComponent.setEnabled( false );
```

The `setEnabled` method enables or disables a component. Using `true` for the argument will enable the component. Using `false` will disable the component.

```
someComponent.requestFocus();
```

The `requestFocus` method tells a component that it should try to become the active component within its window. When a user enters information using the keyboard it will be directed to a `JTextField` or `JTextArea` only if that component has the focus. A program's user can also change which component has the focus by clicking the mouse.

Methods Applicable to Components that Display Text

The `setText` and `getText` methods can be used with four types of components that display text — `JTextFields`, `JTextAreas`, `JLabels`, and `JButtons`.

```
someJTextField.setText( someString );
someJTextArea.setText( someString );
someJLabel.setText( someString );
someJButton.setText( someString );
```

The text displayed by the component is replaced by the text provided as the method's argument.

```
someJTextField.getText()
someJTextArea.getText()
someJLabel.getText()
someJButton.getText()
```

The text currently displayed by the component is returned as the result of an invocation of `getText`.

Methods Applicable to `JTextFields` and `JTextAreas`

There are many methods that can be used with either `JTextFields` or `JTextAreas` (but not with `JLabels` or `JButtons`).

```
someJTextField.setEditable( true );
someJTextField.setEditable( false );
someJTextArea.setEditable( true );
someJTextArea.setEditable( false );
```

The `setEditable` method determines whether or not the person running a program is allowed to use the keyboard and mouse to change the text displayed in a `JTextArea` or `JTextField`. An argument value of `true` makes editing possible. Using `false` as the argument prevents editing.

`someJTextField.setCaretPosition( someInt );` `someJTextArea.setCaretPosition( someInt );`	This method is used to change the position of the text insertion point within the text displayed by the component. Positions within the text are numbered starting at 0.
`someJTextField.getCaretPosition()` `someJTextArea.getCaretPosition()`	This accessor method is used to determine the current position of the text insertion point within the text displayed by the component. Positions within the text are numbered starting at 0.
`someJTextField.getSelectedText()` `someJTextArea.getSelectedText()`	This method returns a `String` containing the text that is currently selected within the text area or text field.
`someJTextField.selectAll();` `someJTextArea.selectAll();`	This method causes all the text currently within the component to become selected.
`someJTextField.select( start, end );` `someJTextArea.select( start, end );`	This method causes the text between the character that appears at the position `start` and the character that appears at position `end` to become selected. If the start or end value is inappropriate, invoking this method has no effect. The argument values must be `int`egers.

The `append` Method

There is one important method that can only be applied to `JTextArea`s.

`someJTextArea.append( someString );`	The contents of `someString` are added after the text currently displayed in the text area.

Methods Associated with `JComboBox`es

Many methods are available for adding, removing, and selecting menu items. In our prototypes for these methods, we will continue to pretend that only `String` values can be used as menu items. In some of the method descriptions, however, we provide a few clues about how a `JComboBox` behaves if menu items that are not `String`s are used.

`someJComboBox.addItem( someString );`	The contents of `someString` are added as a new menu item. In fact, as we hinted earlier, the argument does not need to be a `String`. If an argument of some other type is provided, the text obtained by applying the `toString` method to the argument will appear in the menu.
`someJComboBox.addItemAt( someString, position );`	The contents of `someString` are added as a new menu item at the position specified in the menu. As explained in the description of `addItem`, the first parameter does not actually have to be a `String`.
`someJComboBox.getSelectedItem()`	Returns the menu item that is currently selected. Since menu items other than text strings are allowed, the result of this method might not be a `String`. As a consequence, it is usually necessary to immediately apply the `toString` method to the result of invoking `getSelectedItem`.
`someJComboBox.getSelectedIndex()`	Returns the position of the currently selected menu item. Menu items are numbered starting at 0.
`someJComboBox.getItemCount()`	Returns the number of items currently in the menu.
`someJComboBox.getItemAt( someInt )`	Returns the menu item found in the specified position within the menu. The position argument must be an integer. Menu items are numbered starting at 0. Since items other than text strings are allowed, it is usually necessary to immediately apply the `toString` method to the result of invoking `getItemAt`.
`someJComboBox.setSelectedIndex( position );`	Makes the item at the specified position within the menu become the currently selected item. Menu items are numbered starting at 0.

`someJComboBox.setSelectedItem( someString );`	Makes the item that matches the argument provided become the currently selected item. If no match is found, the selected item is not changed.
`someJComboBox.removeAllItems();`	Removes all the items from a menu.
`someJComboBox.removeItem( someString );`	Removes the item that matches the argument provided from the menu.
`someJComboBox.removeItemAt( position );`	Removes the item at the position specified from the menu. Menu items are numbered starting at 0.

`JPanel` Methods

The methods associated with `JPanel`s can be applied either to a `JPanel` explicitly constructed by your program or to the `contentPane`.

`someJPanel.add( someComponent );` `someJPanel.add( someComponent, position );`	Add the component to those displayed in the panel. If a second argument is included it should be an integer specifying the position at which the new component should be placed. For example a `position` of 0 would place the new component before all others. If no second argument is included, the new component is placed after all existing components in the panel.
`someJPanel.remove( someComponent );`	Remove the specified component from the panel.

3.4 Identifying Event Sources

Sometimes, it is necessary for a program not only to be able to tell that some button has been clicked, but also to determine exactly which button has been clicked. Consider the program whose interface is shown in Figure 3.11. The program displays 10 buttons labeled with the digits 0 through 9 in a configuration similar to a telephone keypad. The program also displays a text field below the buttons.

We would like to discuss how to construct a program that displays such a keypad and reacts when the keypad buttons are pressed by adding to the text field the digits corresponding to the

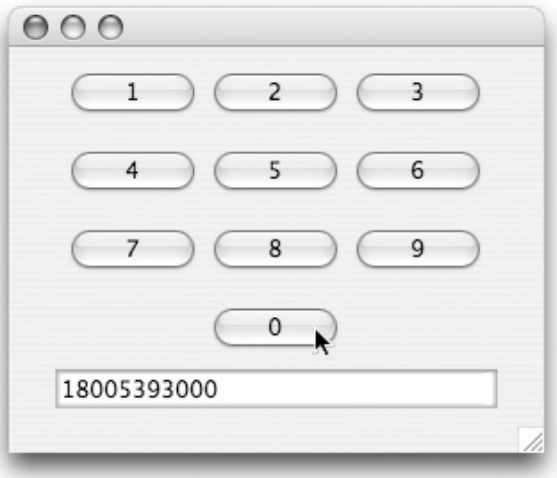

Figure 3.11: A numeric keypad user interface

buttons pressed. To do this, the program somehow has to know both when a button is pressed and be able to determine which of the ten buttons displayed was actually pressed.

We know that we can arrange to have a set of instructions executed every time a button is pressed by placing those instructions in a method definition of the form

```
public void buttonClicked( ) {
    . . .
}
```

When we introduced the notation for defining such methods, we explained what went in the method's body, but we didn't say much about the header. In particular, we never explained the empty pair of parentheses that appears after the name `buttonClicked`.

We have seen that Java uses parentheses to surround the actual parameters in method invocations and constructions. Java borrows this use of parentheses from the mathematical notation for applying functions to values. Thus, when we say "$\sin(\frac{\pi}{2})$", we say $\frac{\pi}{2}$ is being used as an actual parameter to the sine function.

Java also borrows the notion of formal parameter names from the notation used to describe functions in mathematics. For example, a mathematician might define:

$$f(x) = 3x + 2$$

In this case, if asked for the value of $f(7)$, you would hopefully answer 23. In this example, 7 is the argument or actual parameter. The name x is referred to as the *formal parameter*. It is used in the definition as a placeholder for the value of the actual parameter since that value is potentially unknown when the definition is being written. The rules for evaluating an expression like $f(7)$ involve substituting the value 7 for each occurrence of x used in the definition.

The designers of the Squint and Swing libraries realized that the instructions within a `buttonClicked` method might need to know what button was clicked. Therefore the libraries were designed to provide this information to the method when it is invoked. In the Java method header

```
public void buttonClicked( ) {
```

the empty parentheses specify that the method defined expects no parameters. If we define `buttonClicked` in this way, the Java system assumes our particular definition of `buttonClicked` has no need to know which button was clicked. Therefore, the system does not provide this information to our method. If we want the system to tell us which button was clicked, we simply need to add a specification indicating that the method expects a parameter to the method header.

Java is very picky about how we use names in programs. That is why we have to indicate the type of object each name might be associated with when we declare instance variables and local variables. Java treats formal parameter names in the same way. We can't indicate that our method expects a parameter by just including the formal parameter's name in parentheses as we do in mathematical definitions like

$$f(x) = 3x + 2$$

Instead, we have to provide both the name we want to use and the type of actual parameter value with which it will be associated.

The information the system is willing to provide to the `buttonClicked` method is a `JButton`. Therefore if we want to indicate that our `buttonClicked` method will use the name `whichButton` to refer to this information, we would use a header of the form

```
public void buttonClicked( JButton whichButton ) {
```

Just as when declaring a variable name, we are free to choose whatever name we want for a formal parameter as long as it follows the rules for identifiers. So, if we preferred, we might use the header

```
public void buttonClicked( JButton clickedButton ) {
```

Either way, once we include a formal parameter declaration in the header of the method we can use the formal parameter name to refer to the button that was actually clicked.

Knowing this, it is fairly easy to write a version of `buttonClicked` that will add the digit associated with the button that was clicked to a text field. We simply apply `getText` to the button associated with the parameter name and then add the result to the text field. The complete code for a program illustrating how this is done is shown in Figure 3.12. Note that we cannot use the `append` method to add the additional digit. The `append` method is only available when working with a `JTextArea`. Since this program uses a `JTextField`, we must simulate the behavior of `append` by concatenating the current contents of the field, accessed using `getText`, with the digit to be added.

Java is willing to provide information to other event-handling methods in the same way. If the `menuItemSelected` method is defined to expect a `JComboBox` as a parameter, then the formal parameter name specified will be associated with the `JComboBox` in which a new item was just selected before the execution of the method body begins. Similarly, the `buttonClicked` method can be defined to expect a `JButton` as a parameter and the `textEntered` method can be defined to expect a `JTextField` as a parameter. (Note: Entering text in a `JTextArea` does not cause the system to execute the instructions in `textEntered`).

One word of caution, if you define a version of an event-handling method which includes the wrong type of formal parameter declaration, no error will be reported, but the method's code will never be executed. For example, if you included a definition like

```
// A simple implementation of a numeric keypad GUI
public class NumericKeypad extends GUIManager {
   // Change these values to adjust the size of the program's window
   private final int WINDOW_WIDTH = 300, WINDOW_HEIGHT = 200;

   // Width of field used to display digits
   private final int DISPLAY_WIDTH = 20;

   // Used to display the sequence of digits selected
   private JTextField entry;

   // Create and place the keypad buttons in the window
   public NumericKeypad() {
      this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

      contentPane.add( new JButton( "1" ) );
      contentPane.add( new JButton( "2" ) );
      contentPane.add( new JButton( "3" ) );

      contentPane.add( new JButton( "4" ) );
      contentPane.add( new JButton( "5" ) );
      contentPane.add( new JButton( "6" ) );

      contentPane.add( new JButton( "7" ) );
      contentPane.add( new JButton( "8" ) );
      contentPane.add( new JButton( "9" ) );

      contentPane.add( new JButton( "0" ) );

      entry = new JTextField( DISPLAY_WIDTH );
      contentPane.add( entry );
}

   /*
    *  Add the latest digit to the display.
    *     -  parameter "clickedButton" is the button that was clicked
    */
    public void buttonClicked( JButton clickedButton ) {
         entry.setText( entry.getText() + clickedButton.getText() );
    }
}
```

Figure 3.12: Implementation of a numeric keypad

```
public void buttonClicked( JComboBox whichOne ) {
   ...
}
```

the system would essentially ignore the method's definition.

3.5 Using Variables to Remember

So far, we have used variable and parameter names primarily to identify things. In the current version of the keypad program, for example, we use the parameter name `clickedButton` to identify the button that was clicked, and we use the variable name `entry` to identify the text field where the digits should be placed. In this section, we will use the keypad program to explore a more subtle use of variables. We will see that in addition to enabling us to identify things, variables can also help us remember things.

To illustrate this, consider how to add a simple, cosmetic feature to the interface provided by the keypad. In particular, let's think about how to change the program so that the last button clicked is highlighted by displaying the number on that button in orange instead of black.

It is quite easy to make the label of the button that was just clicked turn orange. This can be done by adding the statement

```
clickedButton.setForeground( Color.ORANGE );
```

to the `buttonClicked` method (as long as we also remember to import `java.awt.*`). Unfortunately, if this is all we do, the program won't work quite the way we want. Each time we click a button it will turn orange and then stay orange. Eventually all the buttons will be orange. We only want one button to be orange at a time!

A simple, but inelegant solution would be to associate names with all ten buttons and add ten statements to `buttonClicked` that would make all ten button labels become black just before we set the color for `clickedButton` to be orange. This solution is inelegant because if only one button is orange at any time, then nine of the ten instructions that make buttons turn black are unnecessary. It would be nice if we could just execute a single statement that would turn the one button that was orange back to being black.

To do this, we have to somehow remember which button was clicked last. We can make our program remember this information by associating an instance variable name with this button. That is, we will declare a new variable named `lastButtonClicked` and then add whatever statements are needed to ensure that it always refers to the last button that was clicked. If we do this, then the single statement

```
lastButtonClicked.setForeground( Color.BLACK );
```

can be inserted in the `buttonClicked` method just before the instruction

```
clickedButton.setForeground( Color.ORANGE );
```

Executing these two instructions will first make the old orange button turn black and then make the appropriate new button turn orange as we would like.

The difficult part is determining what statements are needed to ensure that `lastButtonClicked` always refers to the button that is currently orange. If we examine the two lines that we just

suggested adding to the `buttonClicked` method, we can already see a problem. Suppose that the last button that was clicked was the "3" key and now the user has clicked on "7". We would then expect that when the two statements

```
lastButtonClicked.setForeground( Color.BLACK );
clickedButton.setForeground( Color.ORANGE );
```

began to execute, the variable `lastButtonClicked` would be associated with button 3 and `clickedButton` would be associated with button 7. When the statements have finished executing, button 7 will be orange. It is now the "last button clicked". The variable `lastButtonClicked`, however, would still refer to button 3. This is wrong, but easy to fix.

We need to change the meaning of the variable `lastButtonClicked` as soon as we finish executing the two lines shown above. In the example we just considered, we would like `lastButtonClicked` to refer to button 7 after the statements within the `buttonClicked` method are complete. This is the button that is associated with the formal parameter name `clickedButton` during the method's execution. In fact, it will always be the case that after the method finishes, the name `lastButtonClicked` should refer to the button that had been associated with `clickedButton` during the method.

We can accomplish this by adding an assignment statement of the form

```
lastButtonClicked = clickedButton;
```

as the last statement in the method. This statement tells the computer to associate the name `lastButtonClicked` with the same object that is associated with `clickedButton` at the time the statement is executed. Since `clickedButton` always refers to the button being clicked as the method executes, this ensures that after the method's execution is over, the name `lastButtonClicked` will refer to the button that was just clicked. It will then remember this information until the `buttonClicked` method is executed again.

There is one remaining detail we have to resolve. How will this code behave the first time a button is clicked? If the only changes we make to the program are to add a declaration for `lastButtonClicked` and to add the three statements

```
lastButtonClicked.setForeground( Color.BLACK );
clickedButton.setForeground( Color.ORANGE );
lastButtonClicked = clickedButton;
```

to the end of the `buttonClicked` method, then the first time that a button is clicked, no meaning will have been associated with the name `lastButtonClicked`. Although it has been declared, no assignment statement that would give it a meaning would have been executed. As a result, the first time the computer tried to execute the statement

```
lastButtonClicked.setForeground( Color.BLACK );
```

the name `lastButtonClicked` would be meaningless. Unfortunately, Java gets very upset when you try to apply a method using a name that is meaningless. Our program would come to a screeching halt and the computer would display a cryptic error message about a `NullPointerException`.[1]

[1] Although the name `NullPointerException` may appear odd to you at the moment, if you take nothing else out of this section, you should try to remember that an error message containing this word means that you told Java to apply a method using a name to which you had not yet assigned any meaning. This is a fairly common mistake, so knowing how to interpret this error message can be quite helpful. In fact, if you examine the contents of such a message carefully you will find that Java tells you the name of the method within your code that was executing at the time the error occurred and the exact line on which the error was detected.

Later, we will see that Java provides a way to check whether a name is meaningless before using it. At this point, however, there is a simple, safe way to give the name a meaning and make our program function correctly. Clearly there is no way to give the name a meaning that is really consistent with its purpose. Before the first button has been clicked, there is no "correct" button to associate with the name `lastButtonClicked`. If we look at how the name is used, however, we can see that it is easy to give the name a meaning that will lead the program to behave as we desire. All we do with the name `lastButtonClicked` is use it to turn a button black. If we associate it with a button that is already black when the program starts, then using it the first time a button is clicked will just tell a button that is already black to turn black again. For example, we could associate `lastButtonClicked` with button 0 when the program starts. A complete version of the keypad program that uses this technique is shown in Figure 3.13

3.6 Summary

In this chapter, we have pursued two main goals. One obvious goal was to expand your knowledge of the set of methods with which you can manipulate GUI components. A very significant portion of this chapter's text was devoted to the enumeration of available methods and their behaviors.

Our second goal was to solidify your knowledge of some of the most basic mechanisms of the Java language. First, we introduced an entirely new class of methods, *accessor methods*, that provide the ability to get information about the state of an object. Next, we examined the grammatical structure of Java's instructions to distinguish two major types of statements, assignments and invocations. In addition, we identified an important collection of grammatical phrases that are used as sub-components of statements called expressions. We learned that expressions are used to describe the objects and values on which statements operate. Finally, we learned about a new type of name that can be used to refer to objects, *formal parameters*, and saw how these names provide a way to determine which GUI component caused the execution of a particular event-handling method.

```
// A simple implementation of a numeric keypad GUI
public class HighlightedKeypad extends GUIManager {
   private final int WINDOW_WIDTH = 300, WINDOW_HEIGHT = 200;

   // Width of field used to display digits
   private final int DISPLAY_WIDTH = 20;

   // Used to display the sequence of digits selected
   private JTextField entry;

   // Remember which button was clicked last
   private JButton lastButtonClicked;

   // Create and place the keypad buttons in the window
   public HighlightedKeypad() {
      this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

      contentPane.add( new JButton( "1" ) );
      contentPane.add( new JButton( "2" ) );
      contentPane.add( new JButton( "3" ) );
      contentPane.add( new JButton( "4" ) );
      contentPane.add( new JButton( "5" ) );
      contentPane.add( new JButton( "6" ) );
      contentPane.add( new JButton( "7" ) );
      contentPane.add( new JButton( "8" ) );
      contentPane.add( new JButton( "9" ) );

      lastButtonClicked = new JButton( "0" );
      contentPane.add( lastButtonClicked );

      entry = new JTextField( DISPLAY_WIDTH );
      contentPane.add( entry );
   }

   /*
    *  Add the latest digit to the display and highlight its button
    */
    public void buttonClicked( JButton clickedButton ) {
         entry.setText( entry.getText() + clickedButton.getText() );
         lastButtonClicked.setForeground( Color.BLACK );
         clickedButton.setForeground( Color.ORANGE );
         lastButtonClicked = clickedButton;
    }
}
```

Figure 3.13: Using a variable to remember the last button clicked