Chapter 6

Class Action

In Chapter 1, we explained that Java uses the word `class` to refer to

> "A set, collection, group, or configuration containing members regarded as having certain attributes or traits in common."

We have seen that this description applies quite accurately to the classes provided by Java's Swing library. We have constructed groups of `JTextField`s and groups of `JLabel`s as parts of the interface provided by a single program. In each case, the objects created using a given class were distinct from one another but shared many common attributes. Some of the similarities are discernible by just looking at a program's window as it runs. For example, all `JTextField`s look quite similar. Other shared attributes are only visible to the programmer. Different sets of methods are associated with each class. For example, while all `JComboBox`es support the `getSelectedItem` method, this method is not provided for members of the `JTextField` class.

When we first discussed the word `class`, however, we were not talking about library classes. We were explaining why the word `class` appears in the header lines of our sample Java programs. At this point, it should be clear that there is some connection between the library `class`es you have been using and the `class`es you define when you write programs. Primarily, both types of classes involve methods. You define methods within the classes you write and you invoke methods associated with library classes. At the same time, the classes you write seem very different from library classes. You invoke methods associated with library classes, but you don't invoke the `buttonClicked` or `textEntered` methods included in your class definitions. The instructions included in these methods get executed automatically in response to user actions.

In this chapter, we will see that there is actually no fundamental difference between classes you define and the classes provided by the Java libraries. While the programs we have considered thus far have all involved writing only one new class, this is not typical of Java programs. Most Java programs involve the definition of many classes designed just for that program. All Java programs involve one "main" class where execution begins. This class typically constructs `new` objects described by additional class definitions written by the program's author or included in libraries and invokes methods associated with these objects. We will explore the construction of such programs in this chapter.

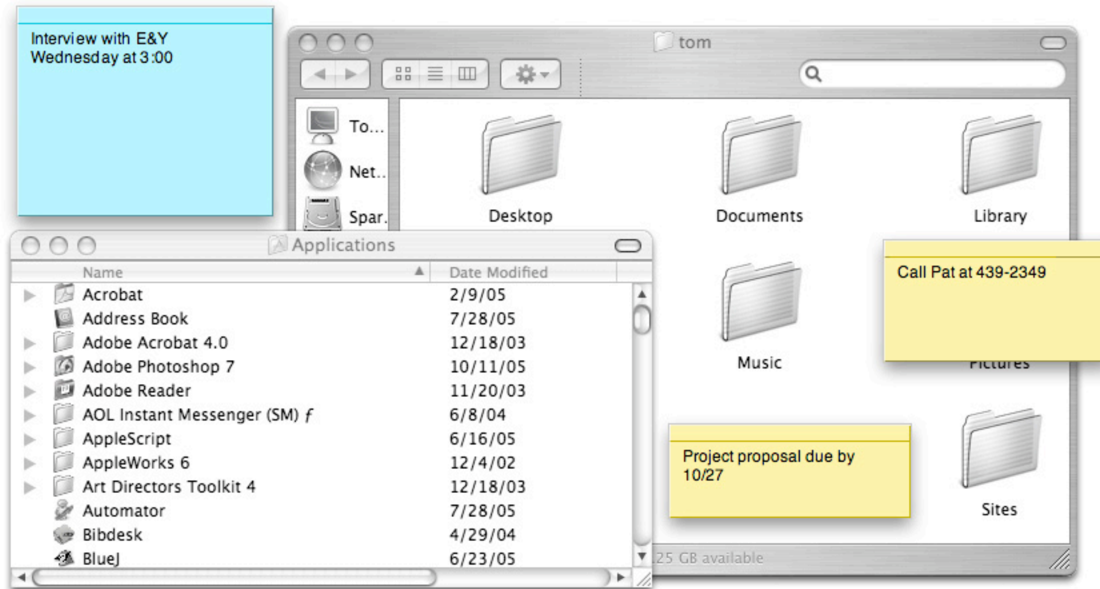


Figure 6.1: Stickies among other windows on a Mac OS system

6.1 Second Class

All the classes we have defined in earlier examples have described the behavior of a single window on the computer's screen. Accordingly, the simplest way we can introduce programs that involve two or more class definitions is to consider how to write programs that display several distinct windows. It isn't hard to think of examples of such programs. While the main window displayed by a word processor is used to display the document being edited, the program will often display separate "dialog box" windows used to handle interactions like selecting a new file to be opened or specifying a special font to use. In addition, when multiple documents are opened, each is displayed in a separate window.

A word processor is a bit too complicated to present here, so we will instead examine a version of what is probably the simplest, useful, multi-window program one could imagine. The program we have in mind is distributed under the name Stickies under Apple's Mac OS X system. In addition, several shareware versions of the program are available for Windows. The goal of the program is to provide a replacement for the handy sticky notes marketed as Post-its®. Basically, all that the program does is enable you to easily create little windows on your computer's screen in which you can type notes to remind yourself of various things. An example of what some of these little windows might look like is shown in Figure 6.1.

We already know enough to write a program that would create a *single* window in which a user could type a short reminder. All we would need to do is place a JTextArea in a program's window. We show an example of how the window created by such a program might look in Figure 6.2 and the code for the program in Figure 6.3.

The people who wrote the Stickies program were obviously afraid that they would be sued by the 3M company if they named their program "Post-its®," and we are afraid that the people who wrote Stickies might sue us if we named our version of this program "Stickies." Accordingly, we

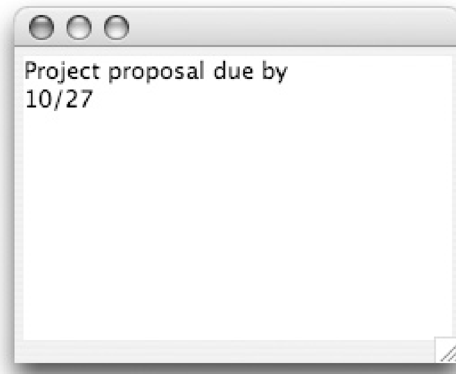


Figure 6.2: A single reminder window created by running the program shown in Figure 6.3

have given our class the clever name `Reminder`. Its code is rather trivial. Its constructor simply creates a window on the screen and then places an empty `JTextArea` in the window.

When running the actual Stickies program, you can create multiple reminder windows by selecting "New Note" from the File menu. We haven't discussed how to create programs with File menus. We can, however, provide an interface that will enable a user to create multiple reminder windows. In particular, what we can do is write another program named `ReminderMaker` that displays a window like the one shown in Figure 6.4. Then, we will place code in the `buttonClicked` method of the `ReminderMaker` program so that each time the button is clicked a new reminder window of the form shown in Figure 6.2 will appear on the screen.

It is surprisingly simple to write the code that will make new reminder windows appear. We have seen many examples where we have told the computer to create a new instance of a library class by using a construction such as

```
new JButton( "Click Here" );
```

In all the constructions we have seen, the name following the word `new` has been the name of a library class. It is also possible, however, to construct an instance of a class we have defined ourselves. Therefore, we can construct a new reminder window by executing a construction of the form

```
new Reminder()
```

Based on these observations, the code for the `ReminderMaker` program is shown in Figure 6.5. This is a very simple class, but it is also our first example of a class whose definition depends on another class that is not part of a standard library. The `ReminderMaker` class depends on our `Reminder` class. Thus, in some sense, we should not consider either of these classes to be a program by itself. In this case, it is the definition of the two classes together that form the program we wanted to write.

In most integrated development environments, when a program is composed of several classes, the text of each class is saved in a separate file and all of these files are grouped together as a single project. In our overview of IDEs in Section 1.4, we showed that after creating a project, we could add a class definition to the project by either clicking on a "New Class" button or selecting a "New

```
// Class Reminder - Creates a window in which you can type a reminder
public class Reminder extends GUIManager {

   // The initial size of the windows
   private final int WINDOW_WIDTH = 250, WINDOW_HEIGHT = 200;

   // The size of the JTextArea
   private final int TEXT_WIDTH = 20, TEXT_HEIGHT = 10;

   // Create a window in which user can type text
   public Reminder() {
      this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );
      contentPane.add( new JTextArea( TEXT_HEIGHT, TEXT_WIDTH ) );
   }
}
```

Figure 6.3: A simple `Reminder` class definition

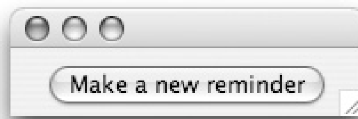


Figure 6.4: A window providing the means to create reminders

```
// Class ReminderMaker - Allows user to create windows to hold brief reminders
public class ReminderMaker extends GUIManager {

    // The size of the program's window
    private final int WINDOW_WIDTH  = 200, WINDOW_HEIGHT = 60;

    // Add the button to the program window
    public ReminderMaker() {
      this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );
      contentPane.add( new JButton( "Make a new reminder" ) );
    }

    // Create a new reminder window
    public void buttonClicked( ) {
        new Reminder();
    }
}
```

Figure 6.5: Definition of the `ReminderMaker` class

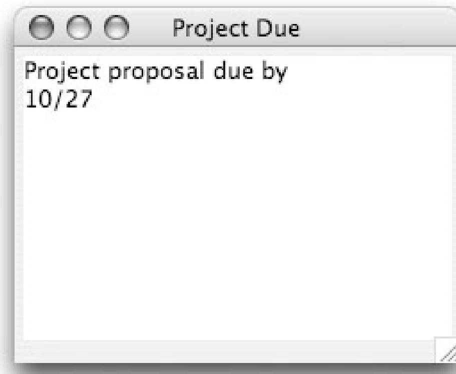


Figure 6.6: A Reminder window with a summary in the title bar

Class" menu item. For a program that involved multiple user-defined classes, we can simply create several classes in this way. The IDE will then provide separate windows in which we can edit the definitions of these classes.

6.2 Constructor Parameters

In most of the constructions we have used, we have included actual parameters that specify properties of the objects we want to create. For example, in our `ReminderMaker` class we use the construction

```
new JButton( "Make a new reminder" )
```

rather than

```
new JButton( )
```

In the construction that creates new `Reminder` objects, on the other hand, we have not included any actual parameters. In this section, we will add some additional features to our `Reminder` program to illustrate how to define classes with constructors that expect and use parameter values.

The first feature we will add is the ability to place a short summary of each reminder in the title bar of the window that displays the complete description. With this change, the windows created to hold reminders will look like the window shown in Figure 6.6 rather than like that shown in Figure 6.2.

There are a few things we have to consider before we can actually change our `Reminder` class to include this feature. In the first place, we have to learn how to place text in the title bar of a window. Then, we have to revise the `ReminderMaker` class to provide a way for the user to enter the summary that should be displayed in the title bar of each reminder. We will address both of these concerns together by redesigning the `ReminderMaker` class so that it both provides a way to enter a summary and displays a title in its own window.

We will associate new names with the two classes we present in this section to avoid confusing them with the very similar classes discussed in the preceding section. We will call the revised classes `TitledReminder` and `TitledReminderMaker`.

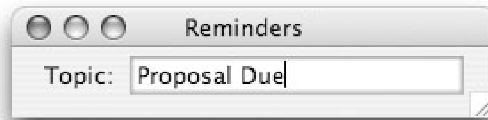


Figure 6.7: Revised interface for creating reminders with titles

Within a `TitledReminderMaker` window, we will place a `JTextField` that will be used to enter a topic for each reminder window to be created. We will also get rid of the "Make a new reminder" button. Instead of creating a new reminder each time a button is pressed, we will create a new reminder whenever the user presses return after entering a summary. As a result, in the `TitledReminderMaker` class, new reminders will be created in the `textEntered` method rather than the `buttonClicked` method. An example of this new interface is shown in Figure 6.7.

If you look carefully at Figure 6.7, you will notice that we have added the title "Reminders" to the title bar of the `ReminderMaker` window. This is actually quite easy to do. The `createWindow` method accepts a title to place in the new window's title bar as a third parameter. Therefore, we can simply add the desired title as a third parameter to the invocation of `createWindow` as shown below:

```
this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT, "Reminders" );
```

The only remaining change to the original `ReminderMaker` class is that we want this new code to pass the text of the desired title as a parameter when it constructs a new reminder. We know that the text to be used will be entered in the `JTextField` provided in the program's interface. Assuming that we associate the name `topic` with this text field, we can pass the text to the reminder constructor by saying

```
new TitledReminder( topic.getText() );
```

This will require that we design the `TitledReminder` class so that it expects and uses the actual parameter information. We will discuss those changes in a moment. Meanwhile, the complete code for the revised `TitledReminderMaker` class is shown in Figure 6.8.

We showed in Section 3.4 that by including a formal parameter name in the declaration of a method, we can inform Java that we expect extra information to be provided when the method executes and that we want the formal parameter name specified to be associated with that information. For example, in the definition of `buttonClicked` shown below,

```
public void buttonClicked( JButton clickedButton ) {
     entry.setText( entry.getText() + clickedButton.getText() );
}
```

(which we originally presented in Figure 3.12) we inform Java that we expect the system to tell us which button was clicked and that we want the formal parameter name `clickedButton` associated with that button.

Formal parameter names can be used in a similar way in the definitions of constructors. If we use the following header when we describe the constructor for the `TitledReminder` class

```
 /*
  * Class TitledReminderMaker - Make windows to hold reminders
  */
public class TitledReminderMaker extends GUIManager {

    // The size of the program's window
    private final int WINDOW_WIDTH = 270, WINDOW_HEIGHT = 60;

    // Size of field used to describe a reminder
    private final int TOPIC_WIDTH = 15;

    // Used to enter description of a new reminder
    private JTextField topic;

    // Add the GUI controls to the program window
    public TitledReminderMaker() {

      this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT, "Reminders" );

      contentPane.add( new JLabel( "Topic: ") );
      topic = new JTextField( TOPIC_WIDTH );
      contentPane.add( topic );
    }

    // Create a new reminder window when a topic is entered
    public void textEntered( ) {
        new TitledReminder( topic.getText() );
    }
}
```

Figure 6.8: Code for the `TitledReminderMaker` class

```
public TitledReminder( String titleLabel )
```

we inform Java that we expect any construction of the form

```
new TitledReminder( . . . )
```

to include an actual parameter expression that describes a string, and that the string passed should be associated with the name `titleLabel` while the instructions in the body of the constructor are executed. In particular, we want to use the `String` passed to the constructor to specify a window title when we invoke `createWindow`. With this in mind, the definition for the constructor of the `TitledReminder` class would look like:

```
// Create a window in which user can type text
public TitledReminder( String titleLabel ) {
  this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT, titleLabel );
  contentPane.add( new JTextArea( TEXT_HEIGHT, TEXT_WIDTH ) );
}
```

The only other difference between `Reminder` and `TitledReminder` would be the class name.

6.2.1 Parameter Correspondence

We have used actual parameters in many previous examples. We have passed actual parameters in constructions like

```
new JTextField( 15 )
```

and in method invocations like

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

In all the previous examples in which we have used actual parameters, the parameters have been passed to methods or constructors defined as part of Squint or the standard Java libraries. We have learned that there are restrictions on the kinds of actual parameters we can pass to each such method. For example, we can pass a color to the `setForeground` method and a number to the `JTextField` constructor but not the other way around. If we pass the wrong type of actual parameter to a library method or constructor, our IDE will produce an error message when we try to compile the program.

Now we can begin to understand why our parameter passing options have been limited. Java requires that the actual parameter values passed in an invocation or construction match the formal parameter declarations included in definitions of the corresponding methods or constructors. If we are using a method or constructor defined as part of Squint, Swing or any other library, we are constrained to provide the types of actual parameters specified by the authors of the code in the library.

Passing the title to be placed in a window's title bar as an actual parameter to the `TitledReminder` constructor is the first time that we have both passed an actual parameter and declared the formal parameter with which it would be associated. As a result, this is the first example where we can see that the limitations on the types of actual parameters we can pass are a result of the details of formal parameter declarations.

Formal parameter declarations not only determine the types of actual parameters we can pass, they also determine the number of parameters expected. We have seen that some constructors expect multiple parameters. In particular, when we construct a `JTextArea` we have provided two actual parameters specifying the desired height and width of the text area. We can explore the definition of constructors that expect multiple parameters by adding another feature to our reminders program.

The Stickies program installed on my computer provides a menu that can be used to select colors for the windows it creates. This may not be obvious when you look at Figure 6.1 if you are reading a copy of this text printed in grayscale, but on my computer's screen I get to have green stickies, pink stickies, and purple stickies. When I first create a new window it appears in a default color. If I want, I can change the default, or I can change an individual window's color after it has been created. We can easily add similar features to our Reminders program.

Once again, since we will be working with revised versions of the two classes that comprise our program, we will given them new names. We will call the new classes `ColorfulReminder` and `ColorfulReminderMaker`. When we create a `ColorfulReminder` we will want to specify both a string to be placed in the window's title bar and the color we would like to be used when drawing the window. That is, it should be possible to create a `ColorfulReminder` using a construction of the form

```
new ColorfulReminder( "Proposal Due!", Color.RED )
```

This construction involves two actual parameters. The first is a `String`. The second a `Color`. Accordingly, in the header for the constructor for the `ColorfulReminder` class we will need to include two formal parameter declarations. We declare these parameters just as we would declare single formal parameters, except we separate the declarations from one another with a comma. Therefore, if we decided to use the names `titleLabel` and `shade` for the parameters, the constructor would look like:

```
public ColorfulReminder( String titleLabel, Color shade ) {
  // Create window to hold a reminder
  this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT, titleLabel );

  JTextArea body;
  body = new JTextArea( TEXT_HEIGHT, TEXT_WIDTH );
  contentPane.add( body );

  // Set the colors of both the text area and the window that contains it
  body.setBackground( shade );
  contentPane.setBackground( shade );
}
```

When multiple formal parameter declarations are included, they must be listed in the same order that the corresponding actual parameters will be listed. That is, since we placed the declaration of `titleLabel` before the declaration of `shade`, we must place the argument `"Proposal Due!"` before `Color.RED` . If we had used the header

```
public ColorfulReminder( Color shade, String titleLabel )
```

then the construction

```
new ColorfulReminder( "Proposal Due!", Color.RED )
```

would be considered an error since the first actual parameter provided is a string while the first formal parameter declared indicates a `Color` is expected.

Of course, if we can define a constructor that expects two parameters, it is also possible to define a constructor that expects even more parameters. For example, if we wanted to make it possible to change the initial size of reminder windows, we might define a class with the constructor

```
public FlexibleReminder( int width, int height,
                         String titleLabel, Color shade ) {
  // Create window to hold a reminder
  this.createWindow( width, height, titleLabel );

  JTextArea body;
  body = new JTextArea( TEXT_HEIGHT, TEXT_WIDTH );
  contentPane.add( body );

  // Set the colors of both the text area and the window that contains it
  body.setBackground( shade );
  contentPane.setBackground( shade );
}
```

In this case, we could use a construction of the form

```
new FlexibleReminder( 400, 300, "Proposal Due", Color.RED )
```

to construct a reminder window.

6.2.2 Choosing `Colors`

Including a `Color` as one of the formal parameters expected by the `ColorfulReminder` class makes it possible to set the color of a reminder window when it is first created. As mentioned above, however, it should also be possible to change a window's color after it has been created. We will explore how to add this feature in the next section. To prepare for this discussion, we will conclude this section by introducing an additional feature of Java's Swing library, a Swing mechanism that makes it easy to let a program's user select a color.

Swing includes a method that can be used to easily display a color selection dialog box. A sample of this type of dialog box is shown in Figure 6.9. Understanding this figure will require a bit of imagination if you are reading a grayscale copy of this text. The small squares displayed in the grid in the middle of the window are all squares of different colors. The mouse cursor in the figure is pointing at a square that is actually bright yellow. A user can select a color by clicking on one of the squares and then clicking on the "OK" button.

The method that produces this dialog box is named `JColorChooser.showDialog`. It expects three parameters: the name of the program's `GUIManager`, typically `this`, a string to be displayed as instructions to the user, and the default color to select if the user just clicks "OK". An invocation of this method might therefore look like:

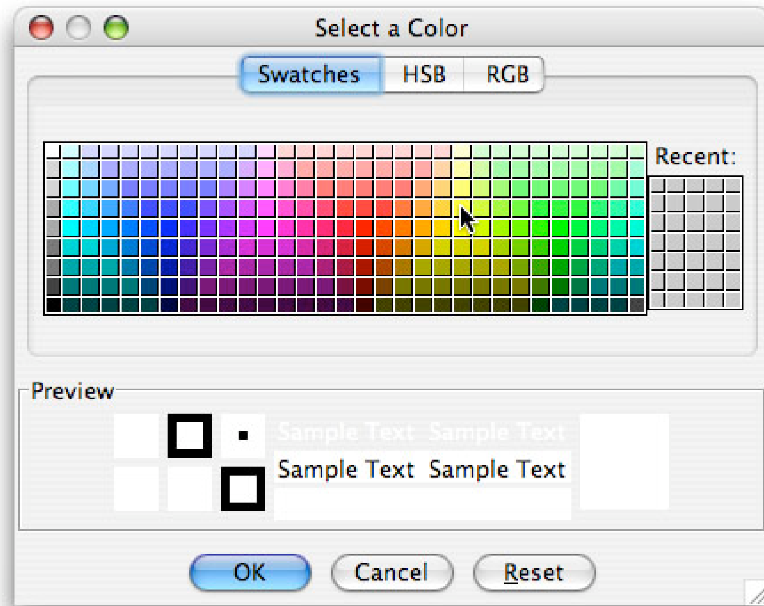


Figure 6.9: Dialog box displayed by the `JColorChooser.showDialog(...)` method

```
JColorChooser.showDialog( this, "Select a Color", Color.WHITE )
```

`JColorChooser.showDialog` is an accessor method. It returns the color the user selected as its result.

As an example of the use of `JColorChooser.showDialog`, the definition of a class designed to work with the `ColorfulReminder` class discussed above is presented in Figure 6.10. The user interface this program provides is shown in Figure 6.11. Like the `TitledReminderMaker` class, this user interface allows a user to create a new reminder window by typing the topic for the reminder in a `JTextField` and then pressing return. It also provides a button the user can press to select a new color for the reminder windows. It uses a variable named `backgroundColor` to remember the color that should be used for the next reminder window created. This variable is initially associated with the color white. When the user presses the "Pick a Color" button, the variable is associated with whatever color the user selects using the dialog box. Each time a new reminder window is created, the current value of `backgroundColor` is passed as an actual parameter to the `ColorfulReminder` constructor. Therefore, once the user has selected a color, it will be used for all windows created until a new color is chosen.

6.3 Method Madness

While the revisions we have made to our reminder program make it much more colorful, the program still doesn't provide as much flexibility as the Stickies program that inspired it. Our program provides no way to change a window's color once it has been created. The Stickies program, on the other hand, lets you change the color of a window even after it has been created. We will explore how to add such flexibility to our program as a means of introducing a very important aspect of

```
/*
 * Class ColorfulReminderMaker - Make windows to hold reminders
 */
public class ColorfulReminderMaker extends GUIManager {
    // The size of the program's window
    private final int WINDOW_WIDTH = 270, WINDOW_HEIGHT = 100;

    // Size of field used to describe a reminder
    private final int TOPIC_WIDTH = 15;

    // Used to enter description of a new reminder
    private JTextField topic;

    // The color to use for the background of the next reminder
    private Color backgroundColor = Color.WHITE;

    // Add the GUI controls to the program window
    public ColorfulReminderMaker() {

      this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT, "Reminders" );

      contentPane.add( new JLabel( "Topic: " ) );
      topic = new JTextField( TOPIC_WIDTH );
      contentPane.add( topic );
      contentPane.add( new JButton( "Pick a Color" ) );
    }

    // Select a new background color
    public void buttonClicked( ) {
        backgroundColor = JColorChooser.showDialog( this, "Select a Color",
                                                    backgroundColor );
    }

    // Create a new reminder window
    public void textEntered() {
        new ColorfulReminder( topic.getText(), backgroundColor );
    }
}
```

Figure 6.10: Complete code for the ColorfulReminderMaker class

Figure 6.11: User interface of the `ColorfulReminderMaker` program

writing programs composed of multiple classes, the ability to pass information between objects through method invocations.

When using library classes, we have seen that in addition to specifying the details of an object through constructor parameters, we can later modify such details using a method invocation. For example, in the `TouchCounter` program presented in Figure 2.10, we specified the initial text displayed by the label named `message` in its constructor:

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

Our program later changes its contents to display the number of times the user has clicked using the `setText` mutator method:

```
message.setText( "I've been touched " + numberOfClicks + " time(s)" );
```

The ability to change the contents of a `JLabel` is provided through the `setText` method. Similarly, we can provide the ability to change the color of a `ColorfulReminder` by including the definition of an appropriate method within our `ColorfulReminder` class. Until now, we have only defined event-handling methods such as `buttonClicked`, and we have only invoked methods that were defined within the Squint or Swing libraries. It is, however, possible for us to define methods other than event-handling methods and then to invoke these methods. Best of all, the process of defining such methods is very similar to the process of defining an event-handling method.

Like the definition of an event-handling method, the definition of our method to set the color of a reminder will begin with the words `public void` followed by the name of the method and any formal parameter declarations. For this method, we will want just one formal parameter, the `Color` to display in the reminder window. We will use the parameter name `newShade`. Unlike event-handling methods, we get to pick any name for the method that is appropriate. We could choose the name `setBackground` to be consistent with the name of the method provided to set the color of Swing GUI components, or we could instead chose a different name like `setColor` or `changeWindowColor`. In fact, we could use a name like `cuteLittleMethod`, but it would be hard to claim that name was *appropriate*. To illustrate that we do have the freedom to choose the name, we will use the appropriate, but not quite standard name `setColor` for our method. Accordingly, our method header will be

```
public void setColor( Color newShade )
```

A complete class definition incorporating the `setColor` method is shown in Figure 6.12. Once again, we have renamed the class to distinguish it from earlier versions. `ColorableReminder` is

```
/*
 * Class ColorableReminder - A window you can type a message in
 */
public class ColorableReminder extends GUIManager {
    // The size of the reminder window
    private final int WINDOW_WIDTH  = 250, WINDOW_HEIGHT = 200;

    // The size of the JTextArea
    private final int TEXT_WIDTH = 20, TEXT_HEIGHT = 10;

    // Area used to hold text of reminder
    private JTextArea body;

    // Add GUI components and set initial color
    public ColorableReminder( String label, Color shade ) {
        // Create window to hold all the components
        this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT, label );

        body = new JTextArea( TEXT_HEIGHT, TEXT_WIDTH );
        contentPane.add( body );

       // Set the colors of both the text area and the window that contains it
        body.setBackground( shade );
        contentPane.setBackground( shade );
    }

    // Change the color of the window's background
    public void setColor( Color newShade ) {
        body.setBackground( newShade );
        contentPane.setBackground( newShade );
    }
}
```

Figure 6.12: The code of the ColorableReminder class

now its name. When the `setColor` method in this class is invoked, it must set the background color of both the `JTextArea` and the `contentPane` so that the whole window will display a single color. Accordingly, the name `body`, which was previously declared as a local variable within the constructor must now be declared as an instance variable. Unsurprisingly, the body of the `setColor` method contains two instructions that are nearly identical to the instructions used to set the color in the constructor. The only difference is that in this context the name `newShade` is used as an actual parameter in the invocations of `setBackground`.

To illustrate the use of our new `setColor` method, we will now modify the reminder maker class. In the new version, which we will name `ColorableReminderMaker`, when the user picks a new color, the program will immediately change the color of the last reminder created. The code for this version of the class is shown in Figure 6.13.

To invoke a method, we have to provide both the name of the method and the name of the object to which it should be applied. Therefore, to use `setColor` as we have explained, we have to associate a name with the last reminder created. We do this by declaring an instance variable named `activeReminder`. An assignment within the `textEntered` method associates this name with each new reminder window that is created. Then, in the `buttonClicked` method, we execute the invocation

```
activeReminder.setColor( backgroundColor );
```

to actually change the color of the most recently created window.[1] This invocation tells Java to execute the instructions in the body of our definition of `setColor`, thereby changing the color of the window's background as desired.

In this example, the method we defined expects only a single parameter. It is also possible to define methods that expect multiple parameters. In all cases, as with constructors, Java will insist that the number of actual parameters provided when we invoke a method matches the number of formals declared in the method's definition and that the types of the actual parameters are compatible with the types included in the formal parameter declarations. Java will associate the actual parameter values with the formal parameter names in the order in which they are listed and then execute the instructions in the method's body.

6.4 Analyze This

this (pronoun, pl. these)

1. used to identify a specific person or thing close at hand or being indicated or experienced, as in :

   He soon knew that this was not the place for him.

(From the New Oxford American Dictionary (via Apple's Dictionary program))

Our example program now consists of two classes with very different structures and roles. Both of the methods defined in `ColorableReminderMaker` are event-handling methods while the `setColor` method in `ColorableReminder` is not designed to respond to user events. The `ColorableReminderMaker`

[1] To ensure that we don't try to change the color of the most recently created reminder before any reminders have been created at all, we don't enable the "Pick a Color" button until after a reminder has been created.

```
// Class ColorableReminderMaker - Make windows to hold reminders
public class ColorableReminderMaker extends GUIManager {
   // The size of the program's window
   private final int WINDOW_WIDTH = 270, WINDOW_HEIGHT = 100;

   // Size of field used to describe a reminder
   private final int TOPIC_WIDTH = 15;

   // Used to enter description of a new reminder
   private JTextField topic;

   // Used to change the color of new reminder backgrounds
   private JButton pickColor;

   // The color to use for the background of the next reminder
   private Color backgroundColor = Color.WHITE;

   // The most recent reminder created
   private ColorableReminder activeReminder;

   // Add the GUI controls to the program window
   public ColorableReminderMaker() {
      this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT, "Reminders" );

      contentPane.add( new JLabel( "Topic: " ) );
      topic = new JTextField( TOPIC_WIDTH );
      contentPane.add( topic );
      pickColor = new JButton( "Pick Background Color" );
      pickColor.setEnabled( false );
      contentPane.add( pickColor );
   }

   // Select a new background color for current and future reminders
   public void buttonClicked( JButton which ) {
      backgroundColor = JColorChooser.showDialog( this, "Select a Color",
                                                  backgroundColor );
      activeReminder.setColor( backgroundColor );
   }

   // Create a new reminder window
   public void textEntered() {
      activeReminder = new ColorableReminder( topic.getText(), backgroundColor );
      pickColor.setEnabled( true );
   }
}
```

Figure 6.13: Complete code for the `ColorablelReminderMaker` class

creates ColorableReminders and tells them what to do through method invocations (i.e., when to change color). The ColorableReminderMaker acts as the boss and the ColorableReminders just listen passively.

In the following section, we will see that it is possible for the communications that occur between objects of different classes to be much more interesting. By the end of the next section, both of the classes in our program will include methods designed to handle events and methods like setColor that do not handle events but are instead invoked explicitly by code in the other class. We will begin the construction of this version of the program by making a rather small change to the ColorableReminder class. We will make it talk to itself!

The code in the ColorableReminder constructor is designed to perform three steps:

1. Create an empty window,
2. Place a text area in the window, and
3. Set the background color used.

The setColor method that we added to the class in the last section is designed to perform the third of these steps. Therefore, it should be possible to replace the last two invocations in the constructor with a single invocation of the setColor method. Recall, however, that when we invoke a method we need to provide both the name of the method and the name of the object to which it should be applied. What name should we use within the constructor to tell Java that we want to apply the setColor method to the object being constructed?

The answer to this question is actually apparent if we look at the first line of the constructor's body. In that line we invoke the createWindow object using the name this. By writing such an invocation, we are telling Java that we want the object being created to create a window for itself. That is, we are applying the createWindow method to the object being constructed. The name this can be used in a constructor to refer to the object being constructed or within a method to refer to the object to which the method is being applied. Thus, we can replace the last two lines of the ColorableReminder constructor with the invocation

```
this.setColor( shade );
```

6.5 Talking Back

The name this can also be used as a parameter in a construction or method invocation. When this is done, the object executing the code containing the construction or invocation is passed as an actual parameter. This provides a way for one object to identify itself to another, making it possible for the other object to communicate with it later.

To illustrate how the name this can be used to facilitate such two-way communications, we will modify our Reminder program to make it even more like the Stickies program. Our program currently only provides a way to change the color of the most recent reminder created. On the other hand, it is possible to change the color of any window created by the Stickies program at any time. The interface provided by Stickies is quite simple. When the user clicks on any window, it becomes the active window. When the user selects a color from the color menu, the program changes the color of the active window rather than of the most recently created window.

Obviously, implementing this behavior will require some modifications to our example classes. Accordingly, we will once again give the new versions new names. This time we will call them `ReactiveReminder` and `ReactiveReminderMaker`.

Like the Stickies program, our program already has some notion of an "active window". If you create several reminder windows and then start typing text, the text you type will appear in one of your reminder windows. If you want to change which window your text appears in, you merely have to click on a different window. After you click, text you type will be inserted in the window you just clicked on. That window is now the active window. In the terminology of Java's Swing library, we say that the window you clicked on has gained the *focus*.

There is an event handling method that you can define if you want to take some special action when your window gains the focus. The method is named `focusGained`. If you include a method definition of the form

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

within a class that extends `GUIManager` then the instructions in the body of the method will be executed whenever one of the GUI components in the `GUIManager`'s content pane gains the focus. When you click on a reminder window, the `JTextArea` in the window gains the focus. Therefore, if we include a definition of `focusGained` in our reminder class, it will be executed when the user clicks on a reminder window.

This is a start. It means that it is possible for a reminder window to become aware that it has become the active window. Unfortunately, this is not enough. To change the color of a reminder window, the user will click on the "Pick a Color" button in the reminder maker window. To actually change a reminder's color, the reminder maker needs to invoke `setColor` on the active window. Therefore, it isn't enough for a reminder to know that it is active. Somehow, a reminder needs a way to tell the reminder maker that it has become the active window.

One object can provide information to another object by invoking one of the methods associated with the other object and passing the information as an actual parameter. We have already seen the reminder maker pass information to a reminder by invoking `setColor`. Now, we have to define a method in the reminder maker that a reminder can invoke to tell the reminder maker that it has become the active window. We will call this method `reminderActivated`. We will invoke this method from the body of the `focusGained` method in the reminder class. It will expect to be passed the window that has become active as a parameter when it is invoked. It will simply associate the name `activeReminder` with this parameter so that the reminder maker can "remember" which window is now active. Therefore, we will define `reminderActivated` as follows:

```
// Notice when the active reminder window is changed
public void reminderActivated( ReactiveReminder whichReminder ) {
    activeReminder = whichReminder;
}
```

We know that the invocation of `reminderActivated` that we place in the reminder class must look something like:

```
x.reminderActivated( y );
```

The interesting part is deciding what to actually use where we have written "x" and "y".

Here "y" represents the actual parameter that should be passed when `reminderActivated` is invoked. We said that we needed to pass the reminder that was becoming active to the reminder maker. This is the same as the window that will be executing the `focusGained` method. Therefore, we can use the name `this` to refer to the window. Now we know that our invocation must look like

```
x.reminderActivated( this );
```

We will also use `this` to determine the correct replacement for "x". The problem is that whatever we use for "x" should refer to the reminder maker, not a reminder window. If we use the name `this` within the `ReactiveReminder` class, it will refer to a reminder window, but if we use it within the `ReactiveReminderMaker` class it will refer to the reminder maker. The solution, then, is to pass `this` from the reminder maker in each reminder construction it executes. If we view the parameters to the constructor as a message sent to the new object, including `this` in the list is like putting a return address on the envelope. It tells the new object who created it.

To accomplish this, we first have to both add `this` as an actual parameter in the construction:

```
activeReminder = new ReactiveReminder( topic.getText(), backgroundColor,
                                       this );
```

and add a formal parameter declaration that associates a name with the extra parameter in the header of the constructor for the `ReactiveReminder` class;

```
public ReactiveReminder( String label, Color shade,
                         ReactiveReminderMaker creator ) {
```

At this point, we are close, but not quite done. The name `creator` can now be used to refer to the reminder maker within the reminder constructor. Formal parameter names, however, can only be used locally within the constructor or method with which they are associated. Therefore, we cannot use the name `creator` to refer to the reminder maker within the body of `focusGained`.

To share information between a constructor and a method (or between two distinct invocations of methods) we must take the information and associate it with an instance variable. We will do this by declaring an instance variable named `theCreator` and associating it with the reminder maker by executing the assignment

```
theCreator = creator;
```

within the body of the constructor. Then, we can place the invocation

```
theCreator.reminderActivated( this );
```

in the `focusGained` method. A complete copy of the revised classes can be found in Figures 6.14 and 6.15.

Both of the classes in this version of the program include methods that are designed to handle GUI events and other methods that are invoked explicitly within the program itself. `ReactiveReminder` defines the event-handling method `focusGained` in addition to the simple mutator method `setColor`. The `ReactiveReminderMaker` class included two methods that handle GUI events, `textEntered` and `buttonClicked`, in addition to the `reminderActivated` method which is invoked by code in the `ReactiveReminder` class.

```
/*
 * Class ReactiveReminder - A window you can type a message in
 */
public class ReactiveReminder extends GUIManager {
    // The size of the reminder window
    private final int WINDOW_WIDTH  = 250, WINDOW_HEIGHT = 200;

    // The size of the JTextArea
    private final int TEXT_WIDTH = 20, TEXT_HEIGHT = 10;

    // Area used to hold text of reminder
    private JTextArea body = new JTextArea( TEXT_HEIGHT, TEXT_WIDTH );

    // The ReactiveReminderMaker that created this reminder
    private ReactiveReminderMaker theCreator;

    // Add GUI components and set initial color
    public ReactiveReminder( String label, Color shade,
                             ReactiveReminderMaker creator ) {
        // Create window to hold all the components
        this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT, label );

        theCreator = creator;
        contentPane.add( body );
        this.setColor( shade );
    }

    // Change the color of the window's background
    public void setColor( Color newShade ) {
        body.setBackground( newShade );
        contentPane.setBackground( newShade );
    }

    // Notify the manager if the user clicks on this window to make it active
    public void focusGained() {
        theCreator.reminderActivated( this );
    }
}
```

Figure 6.14: Code for the `ReactiveReminder` class

```
// Class ReactiveReminderMaker - Make windows to hold reminders
public class ReactiveReminderMaker extends GUIManager {
    // The size of the program's window
    private final int WINDOW_WIDTH = 270, WINDOW_HEIGHT = 100;

    // Used to enter description of a new reminder
    private JTextField topic = new JTextField( 15 );

    // Used to change the color used for new reminder backgrounds
    private JButton pickColor = new JButton( "Pick a Color" );

    // The color to use for the background of the next reminder
    private Color backgroundColor = Color.WHITE;

    // The most recent reminder created
    private ReactiveReminder activeReminder;

    // Add the GUI controls to the program window
    public ReactiveReminderMaker() {

      this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT, "Reminders" );

      contentPane.add( new JLabel( "Topic: " ) );
      contentPane.add( topic );
      pickColor.setEnabled( false );
      contentPane.add( pickColor );
    }

    //  Select a new background color for current and future reminders
    public void buttonClicked() {
        backgroundColor = JColorChooser.showDialog( this, "Select a Color",
                                                    backgroundColor );
        activeReminder.setColor( backgroundColor );
    }

    // Create a new reminder window
    public void textEntered() {
        activeReminder = new ReactiveReminder( topic.getText(), backgroundColor,
                                               this );
        pickColor.setEnabled( true );
    }

    // Notice when the active reminder window is changed
    public void reminderActivated( ReactiveReminder whichReminder ) {
        activeReminder = whichReminder;
    }
}
```

Figure 6.15: Code for the ReactiveReminderMaker class

It is worth observing that if you look at just the code for the ReactiveReminderMaker class, the reminderActivated method might easily be mistaken for an event handling method. Like the other two methods in this class, the instructions in its body are designed to make the program react to a user action appropriately. The only reason we distinguish the other two methods from reminderActivated is that buttonClicked and textEntered are executed by some mysterious mechanism in the Java system while reminderActivated is executed when the code we wrote in ReactiveReminder explicitly invokes it. We point this out to make the "mysterious mechanism" less mysterious. Just as we explicitly invoke reminderActivated, it should now be clear that somewhere in the code of the Squint or Swing library there are statements that explicitly invoke textEntered and buttonClicked. There is really no fundamental difference between an event-handling method and any other method we define.

6.6 Defining Accessor Methods

In Chapter 3 we introduced the distinction between mutator methods and accessor methods. Any method whose purpose is to change some aspect of an object's state is classified as a mutator method. For example, because the setText method changes the contents of a GUI component, it is classified as a mutator. Similarly, the setColor method we defined in our reminder class would be classified as a mutator method. Accessor methods serve a different role. An accessor method provides a way to obtain information about an object's state. The getText and getSelectedItem methods are good examples of accessor methods.

The two examples of methods presented so far in this chapter, setColor and reminderActivated, are both examples of mutator methods. One changes a clearly visible property of an object, its color. The other changes the reminder associated with the variable activeReminder within the reminder maker. While this change is not immediately visible, it does determine how the program will react the next time the user presses the "Pick a Color" button.

Our goal in this section is to introduce the features used to define accessor methods. We will use a very simple example. If we have a setColor method in our reminder class, it might be handy to have a getColor method. In fact, introducing such a method will enable us to fix a rather subtle flaw in the current version of our program.

If you look back at the invocation that causes the color selection dialog box to appear:

```
backgroundColor = JColorChooser.showDialog( this, "Select a Color",
                                            backgroundColor );
```

you will notice that most of the actual parameters included in this invocation are rather mysterious. The second parameter is the only obvious one. It determines what prompt the user will see in the dialog box. The first parameter is included because each dialog box created by Swing must be associated with some existing window on the screen. In this context, this refers to the reminder maker window. The last parameter is the one we are really interested in. The color dialog is typically used to change something's color. The third parameter is supposed to specify the color you are starting with. This enables the system to include a patch of the original color in the dialog box so that the user can compare it to the alternative being chosen.

As currently written, our program isn't really using this third parameter correctly. The name backgroundColor in our program will be associated with the background color of the last window whose color was changed. This may not be the same as the background color of the currently active window.

If there was a `getColor` method defined in the `ReactiveReminder` class we could fix this problem quite easily. Since the variable `activeReminder` is always associated with the active window, we could rewrite the invocation that creates the color chooser dialog as:

```
backgroundColor = JColorChooser.showDialog( this, "Select a Color",
                                             activeReminder.getColor());
```

Now, all we have to do is actually define the `getColor` method!

The process of defining an accessor method is very similar to that of defining a mutator method. We must write a header including the name of the method and declarations of any formal parameters it expects. We also write a body containing a list of instructions to be executed when the method is invoked. The difference is that the purpose of the instructions we write will be to determine what information to return as the "answer" to the question being asked when the method is invoked. Java therefore requires that the definition of an accessor method contains two special components. The first describes the type of the information that will be returned as an answer. The second specifies the actual answer to be returned.

The specification of the type of the answer produced by an accessor method appears in its header. In all of the method definitions we have seen, the second word has been `void`. This actually describes the information the method will return. A mutator method returns `void`, that is, nothing. In the definition of an accessor method, we will replace `void` with the name of the type of information the method is designed to produce. This type is called the method's *return type* For example, since `getColor` should obviously produce a `Color`, its header will look like

```
public Color getColor() {
```

In addition, somewhere in the instructions we have to explicitly tell Java what answer our method should return to the code that invoked it. This is done by including an instruction of the form

```
return expression;
```

in the method body. The value or object described by the expression in the `return` statement will be used as the result of the invocation. For example, to return the color of a reminder, we could use the `return` statement

```
return contentPane.getBackground();
```

or

```
return body.getBackground();
```

Therefore, the complete definition of the `setColor` method might be

```
public Color getColor() {
    return contentPane.getBackground();
}
```

When a `return` statement is executed, the computer immediately stops executing instructions in the method's body and returns to executing instructions at the point from which the method was invoked. As a result, a `return` statement is usually included as the last statement in the body

of an accessor method. `return` statements may be included at other points in a method's body, but the last statement executed by an accessor method must be a `return`. In our example, this is not an issue since the `return` statement is actually the only statement in the body. This is because it is very easy to determine the color of a reminder. If a more complicated process that involved many statements was required, all these statements could appear in the body of our method followed by a `return`.

6.7 Invisible Objects

Most of the objects we have considered thus far appear as something concrete and visible on your computer screen. When you say

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

you know that you made the computer construct a new button because you can see it on the screen. It is important to realize, however, that it is possible to create an object that never becomes visible. For example, if you create a button by saying

```
JButton invisibleButton;
invisibleButton = new JButton( "Click Me" );
```

and don't say

```
add( invisibleButton );
```

Java still has to create the button even though you can't see it. Somewhere inside the computer's memory, Java keeps information about the button because it has to remember what the button is supposed to look like in case you eventually do add it to your content pane. For example, if you execute the statement

```
invisibleButton.setText( "Click me if you can" );
```

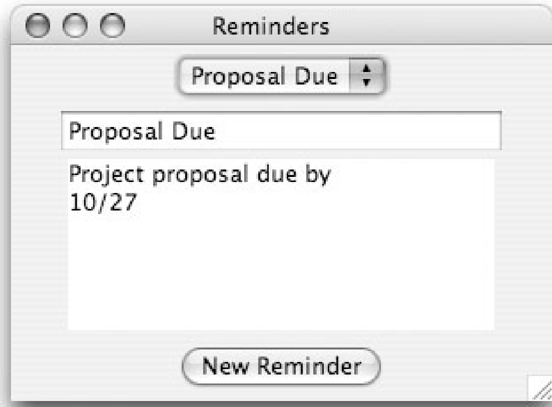
Java has to have some way to record the change even though it doesn't have any visible effect.

Invisible buttons are clearly not very useful. The idea that creating an object forces Java to keep track of information about the object even if it isn't visible, on the other hand, is very useful and important. In this section, we will explore an example in which we define a class of objects that will never be visible on the screen. They will be used to record information that will be displayed, but the objects themselves will remain invisible.

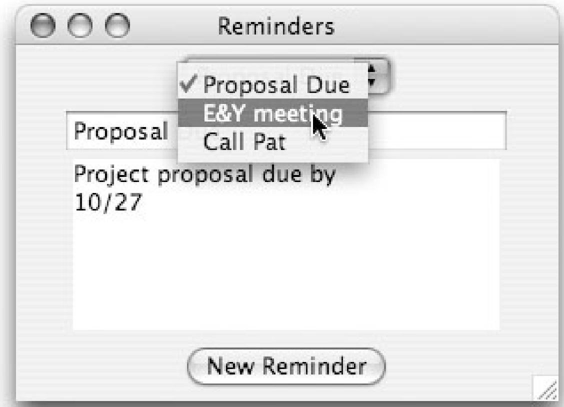
It probably will come as no shock that the example we have in mind is yet another variant of the Reminders program. The good news is that this variant will be very different from the programs we have presented in the preceding sections.

The idea is that a program to keep track of a set of reminders doesn't need to have a separate window to display each reminder. An alternate interface would be to have a single window that could display the contents of one reminder together with a menu from which one could select any of the reminders when one wanted its contents to be displayed. To make this concrete, Figure 6.16 shows how information might be presented by such a program.

The snapshot presented in Figure 6.16(a) shows how the program would look while a user was simply examining one reminder. Figure 6.16(b) shows how the user would request to see the



(a)



(b)

(c)

(d)

Figure 6.16: An alternate interface for viewing reminders

contents of a different reminder. When the mouse is depressed on the menu in the window, a list of the topics of all the reminders would be displayed. The user could select any item in the list. If the user selected the second item as shown in Figure 6.16(b), then, when the mouse was released, the program would display the contents of the selected reminder as shown in Figure 6.16(c). Whenever a given reminder was displayed, the user could edit the contents of the text field and text area displayed in the window to update, correct, or extend the information previously entered. In particular, as shown in Figure 6.16(d), if the user clicked on the "New Reminder" button, the program would display a reminder with a default topic such as "Reminder 4". The user could then edit the topic and contents to include the new reminder in the collection maintained by the program.

In the previous versions of the reminder program, a `Reminder`[2] was actually a window displayed on the screen that contained a text area in which the desired text was displayed. If the program was keeping track of four reminders, there would be four windows displayed.

In this new version of the program, a reminder will not itself correspond to anything visible on the screen. The program displays one text field and one text area. At various times these two components are used to display the text of various reminders, but when we create a new reminder, the program does not create and display a new text area or text field. It just uses `setText` to change what an existing component displays. A reminder is no longer a window or even a GUI component. A reminder is just a pair of strings. Basically, a reminder is a piece of information, not the means used to display it.

It is still very useful, however, to define a `Reminder` class so that we can create objects to represent the contents of various reminders. The definition of such a class is shown in Figure 6.17.

The new class is named `InvisibleReminder`. It differs from all the other classes we have defined so far in a very significant way. Every other class we have seen has included the phrase `extends GUIManager` in its header. This class does not. The phrase `extends GUIManager` indicates that a class definition describes how to respond to GUI components within a window. As we have seen, each time we construct an object of a class that extends `GUIManager`, we end up with a new window. Since this class does not extend `GUIManager`, creating a new `InvisibleReminder` will not create a new window. In fact, it you try to reference the `contentPane` or invoke `this.createWindow(...)` within such a class, your IDE will identify that code as an error.

The fact that we cannot see an `InvisibleReminder` does not mean that it is impossible to write a program that displays its contents. Within the definition of the class, we have included accessor methods that make it possible to determine what the contents of a given reminder are. To construct a program to provide an interface like that shown in Figure 6.16, we would define a second class that would build a GUI interface including a text field and a text area. Within the code of that class, we would associate names with the text field, text area, and the `InvisibleReminder` the user had most recently selected from the menu. This class could then use the accessor methods associated with an `InvisibleReminder` together with the `setText` method to display a reminder. In particular, assuming that the text field is named `topic`, the text area is named `reminder`, and the `InvisibleReminder` is named `currentReminder`, we could display the desired information by executing the instructions

```
topic.setText( currentReminder.getLabel() );
```

[2]Even though we used a variety of names to distinguish various versions of our reminder class from one another, we now use the name of the original version of the class, `Reminder`, to refer to all of the versions since we are discussing properties shared by all the variations of the class we presented.

```java
/*
 * InvisibleReminder --- Holds the contents and summary of a reminder
 */
public class InvisibleReminder {

    // The strings that make up the contents of the reminder
    private String label;
    private String body;

    // Turn a pair of strings into a reminder
    public InvisibleReminder( String myLabel, String myBody ) {
        label = myLabel;
        body = myBody;
    }

    // Get the short summary of a reminder's contents
    public String getLabel() {
        return label;
    }

    // Get the full details of a reminder
    public String getBody() {
        return body;
    }

    // Change the strings that are the contents of a reminder
    public void setContents( String myLabel, String myBody ) {
        label = myLabel;
        body = myBody;
    }
}
```

Figure 6.17: The definition of the `InvisibleReminder` class

```
reminder.setText( currentReminder.getBody() );
```

The only subtle aspect of constructing such a program is arranging to have the `currentReminder` variable associated with the latest menu item selected by the user. While perhaps subtle, this is actually quite easy if we take full advantage of the flexibility provided by the `JComboBox` class defined within Swing.

In all of our previous examples that used `JComboBox`es, all the items we added to the menu were strings. When we discussed `JComboBox`es, however, we explained that Swing would let us add objects other than strings to a menu. That is why Java insists that we say

```
someJComboBox.getSelectedItem().toString()
```

rather than simply saying

```
someJComboBox.getSelectedItem()
```

when we want to get the string a user has selected from a menu. If a menu contains items other than strings,

```
someJComboBox.getSelectedItem()
```

may return something other than a string. In particular, if we use the `addItem` method to place `InvisibleReminder`s in a `JComboBox`, then when we execute

```
someJComboBox.getSelectedItem()
```

the value returned will be an `InvisibleReminder`.

Unfortunately, just as Java will not let us assume that `getSelectedItem` will return a string, it will not let us assume it returns an `InvisibleReminder`. We must tell it explicitly to treat the item selected in the menu as an `InvisibleReminder`. There is no `toInvisibleReminder` method to accomplish this. Instead, there is a form of expression called a *type cast* that provides a general way to tell Java that we want it to assume that the values produced by an expression will have a particular type.

A type cast takes the form

```
( type-name ) expression
```

That is, we simply place the name of the type of value we believe the expression will produce in parentheses before the expression itself. In the case of extracting an `InvisibleReminder` from a menu, we might say

```
currentReminder = (InvisibleReminder) reminderMenu.getSelectedItem();
```

While we no longer use the `toString` method when we invoke `getSelectedItem`, the `toString` method still plays an important role when we want to build a menu out of items that are not `String`s. If we build a menu by adding `InvisibleReminder`s as items to the menu, Java still needs some way to display strings that describe the items in the menu. To make this possible, it assumes that if it applies the `toString` method to any object we add to a `JComboBox`, the result returned will be the string it should display in the menu. Right now, our definition of `InvisibleReminder` does not include a definition of `toString`, so this will not work as we desire. We can fix this by simply adding the defintion

```
public String toString() {
    return this.getLabel();
}
```

to our InvisibleReminder class.

To illustrate these ideas, a complete definition of a ReminderViewer class that would provide an interface like that shown in Figure 6.16 is presented in Figures 6.18 and 6.19.

In all of our previous examples of code, we have avoided the use of initializers in instance variable and local variable declarations. As we explained in Section 2.6, we did this to ensure that you learned to clearly distinguish the roles of declarations and assignments in Java. In this example and the remainder of the examples we present in this text, however, we will assume that you are comfortable enough with the use of declarations and assignments that we can begin to use initializers without confusion. In particular, you will note that in this example we use initializers to create the GUI components named reminderMenu, topic, and reminder.

Begin by looking at the code for the menuItemSelected method in Figure 6.19. When the user selects a new item from the menu, this ensures that the contents of the selected reminder are displayed. As explained above, we want to allow the program's user to update the text of a reminder while it is displayed. Therefore, the first step in this method is designed to make sure any such changes are recorded. It uses the setContents mutator method to replace the previous contents of the reminder with the text found in the text field and text area. Next, the method uses a type cast with the getSelectedItem method to associate the newly selected reminder with the currentReminder variable. Finally, it displays the contents of the reminder using the getLabel and getBody methods.

Once this method's function is understood, it should be quite easy to understand the code in the buttonClicked method. Like menuItemSelected, it begins by saving any updates to the currently displayed reminder. Then it creates a new reminder with a default label of the form "Reminder N". It displays the (not very interesting) initial contents of the reminder in the text field and text area. Then it adds the new reminder to the menu.

The interesting thing about this program is how important the role played by the InvisibleReminder class is even though we never actually see an InvisibleReminder on our screen. In our introduction to Java, we have deliberately focused our attention on classes that describe objects that have a visible presence on the screen. We believe that starting with such objects makes it easier to grasp programming concepts because "seeing is believing." As you learn more about programming, however, you will discover that the most interesting classes defined in a program are often the ones you cannot see. They represent abstract information that is critical to the correct functioning of the program, even though they may not have any simple, visual representation.

6.8 Private Parts

We have already noted that it is possible for the code we place in the body of a method or constructor within a class to invoke another method defined in the same class. We saw in Section 6.4 that we could use the invocation

```
this.setColor( shade );
```

within the constructor of the ColorableReminder class. In some cases, in fact, it is worth writing methods that are only invoked from code in the class in which they are defined. When we are

```
// Class ReminderViewer - Provide menu-based access to a collection
//           of reminder notes through a single window
public class ReminderViewer extends GUIManager {
   // The size of the program's window
   private final int WINDOW_WIDTH = 300, WINDOW_HEIGHT = 200;

   // The size of the JTextArea
   private final int TEXT_WIDTH = 20, TEXT_HEIGHT = 6;

   // Menu used to select reminder to view/edit
   private JComboBox reminderMenu = new JComboBox();

   // Field used to enter, display, and edit reminder topics
   private JTextField topic = new JTextField( TEXT_WIDTH );

   // Text Area used to enter, display and edit reminder messages
   private JTextArea reminder = new JTextArea( TEXT_HEIGHT, TEXT_WIDTH );

   // Count of the number of reminders created so far
   private int reminderCount = 0;

   // The current reminder
   private InvisibleReminder currentReminder;

   // Place button, menu, a summary field, and a text area in a window
   public ReminderViewer() {
        this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT, "Reminders" );

        // Create a new Reminder
        reminderCount = reminderCount + 1;
        currentReminder = new InvisibleReminder( "Reminder "+reminderCount, "" );
        reminderMenu.addItem( currentReminder );

        // Display contents of new reminder
        topic.setText( currentReminder.getLabel() );

        // Add all the GUI components to the display
        contentPane.add( reminderMenu );
        contentPane.add( topic );
        contentPane.add( reminder );

        contentPane.add( new JButton( "New Reminder" ) );
   }
```

Figure 6.18: Variable and constructor for the ReminderViewer class

```
    // Save the current reminder contents and create a new reminder
    public void buttonClicked( ) {
         // Save changes made to current reminder
         currentReminder.setContents( topic.getText(), reminder.getText() );

         // Create a new reminder
         reminderCount = reminderCount + 1;
         currentReminder = new InvisibleReminder( "Reminder "+reminderCount, "" );

         // Display contents of new reminder
         topic.setText( currentReminder.getLabel() );
         reminder.setText( currentReminder.getBody() );

         // Change menu to select new reminder
         reminderMenu.addItem( currentReminder );
         reminderMenu.setSelectedItem( currentReminder );
    }

    // Save the current reminder and display the reminder selected from menu
    public void menuItemSelected() {
         // Save changes made to current reminder
         currentReminder.setContents( topic.getText(), reminder.getText() );

         // Access reminder selected through menu
         currentReminder = (InvisibleReminder) reminderMenu.getSelectedItem();

         // Display conents of new reminder
         topic.setText( currentReminder.getLabel() );
         reminder.setText( currentReminder.getBody() );
    }
}
```

Figure 6.19: Method definitions for the `ReminderViewer` class

trying to understand a large program composed of many classes, it is useful to be able to easily distinguish those parts of a class definition that are relevant to other classes from those that are only of local interest. We do this by using the word `private`. By placing the word `private` before variable declarations in our classes, we are saying that they can only be used within the class in which they are defined. Similarly, if we find that we want to define a method that is only intended for use within its own class, then that method should be declared as `private` rather than `public`.

The `ReminderViewer` class presented in the last section provides a good example to illustrate the usefulness of such `private` methods. If you compare the steps performed by the `buttonClicked` and `menuItemSelected` methods, you will notice that they have a lot in common. They both start with the instruction

```
currentReminder.setContents( topic.getText(), reminder.getText() );
```

and they both end with the statements

```
topic.setText( currentReminder.getLabel() );
reminder.setText( currentReminder.getBody() );
```

It would be nice if we could avoid repeating these statements in two methods. Doing so would obviously save us a little typing time. It might also make our program easier to modify. If we decided later that we wanted to display the contents of a reminder in some different way (perhaps concatenated together in a single text area rather than in two components), we would currently have to change both copies of the instructions at the ends of these methods. Having only one copy to change would make things easier and decrease the likelihood of making errors while incorporating such changes.

The best way to approach the task of eliminating such repeated code is not just to look for repetition, but to try to find a logical, abstract way to describe what the original code is doing. This will often enable one to identify good ways to regroup statements into separate methods.

In this situation, one way to describe what these two methods have in common is that they both seek to replace the currently displayed reminder. The function of the `buttonClicked` method can be summarized as:

1. Create a new reminder

2. **Replace** the currently displayed reminder with the new reminder

The function of the `menuItemSelected` method can be described as

1. Identify the item currently selected in the menu

2. **Replace** the currently displayed reminder with the selected reminder

Apparently, the task of replacing the displayed reminder is a common element of both of these processes. Therefore, it might help to define a private `replaceReminder` method to perform this task. It would be defined to take the new reminder as a parameter so that it would be possible to either pass it a newly created reminder or a reminder selected through the menu.

In Figure 6.20 we provide a definition for `replaceReminder` together with revised versions of `buttonClicked` and `menuItemSelected` that use this `private` method. The code in this figure would serve as a replacement for the code shown in Figure 6.19. You may notice that it isn't clear that eliminating the repeated code has really reduced the amount of typing that would be required to enter this program. Eliminating such repetition, however, is usually worthwhile simply because it can simplify the process of debugging new code and the process of modifying existing code.

```
// Replace the currently displayed reminder with a different reminder
private void replaceReminder( InvisibleReminder replacement ) {
    // Save changes made to current reminder
    currentReminder.setContents( topic.getText(), reminder.getText() );

    currentReminder = replacement;

    // Display contents of new reminder
    topic.setText( currentReminder.getLabel() );
    reminder.setText( currentReminder.getBody() );

    // Change menu to select new reminder
    reminderMenu.setSelectedItem( currentReminder );
}

// Save the current reminder contents and create a new reminder
public void buttonClicked( ) {
    // Create and display a new reminder
    reminderCount = reminderCount + 1;
    InvisibleReminder newReminder =
              new InvisibleReminder( "Reminder " + reminderCount, "" );
    reminderMenu.addItem( newReminder );
    this.replaceReminder( newReminder );
}

// Save the current reminder and display the reminder selected from menu
public void menuItemSelected() {
    // Access reminder selected through menu and display its contents
    this.replaceReminder( (InvisibleReminder) reminderMenu.getSelectedItem() );
}
```

Figure 6.20: Using the `private` method `replaceReminder`

6.9 Summary

In the preceding chapters, much of the functionality of the programs we constructed depended on classes like `JTextArea` and `NetConnection` provided as part of Java libraries. These classes define types of objects that play important roles as components of the programs we have written. In this chapter, we showed that we can define classes of our own to implement components of our programs that do not correspond to types already available in the libraries.

When we use these mechanisms to decompose our program into separate components, it is usually necessary for the components to exchange information as the program executes. This can be accomplished in several ways. The methods and constructors we define can be designed to accept information they need as parameters. We also showed that by associating information received as a parameter value with an instance variable name a method or constructor can make that information directly accessible to other methods within its class.

Accessor methods provide another means for information to flow from one class to another. When an accessor method is defined, it is necessary to specify both the type of information the accessor method will produce in its header and the particular value to be returned using a `return` statement.

While it is often essential to exchange information between classes, another important role of classes is to limit the flow of information. One goal of decomposing our program into separate classes is to make each class as independent of the detailed information maintained by other classes as possible. This can make programs much easier to construct, understand, and maintain. With this in mind, we stressed the importance of `private` components of classes. In general, all variables declared in a class should be private, and any method that is designed only to be used by other methods within its own class should also be private.