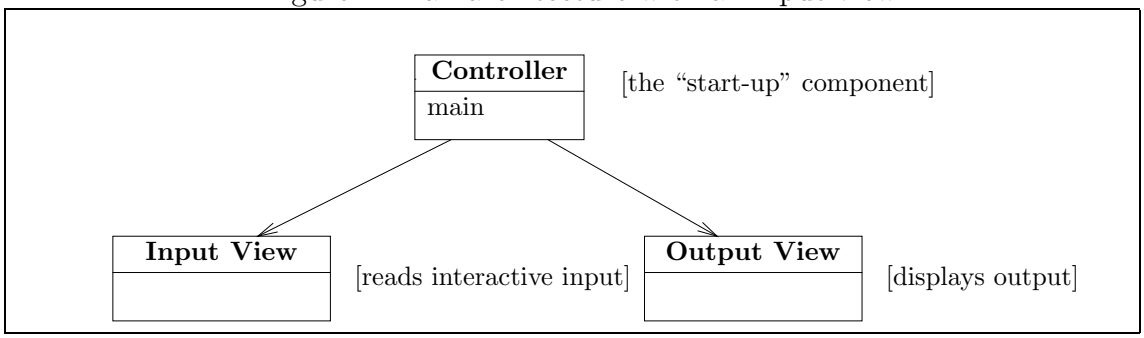
Chapter 4

Input, Output, and State

Realistic programs contain subassemblies that are dedicated to input and output activities. This chapter presents standard techniques for designing and using input-output classes. The objectives are to

- *employ* interactive input*, where an application can interact with its user and request input when needed for computation;*
- *use* inheritance *to design graphics windows that display output in the forms of text, colors, and pictures.*
- *show how an object can "remember" the "state" of the computation by means of* field variables*.*

Figure 4.1: an architecture with an input-view



4.1 Interactive Input

The temperature-conversion program in Figure 8, Chapter 3, receives its input from a *program argument*, which is data that is supplied the moment that a program is started. (Review the section, "Input via Program Arguments," in that chapter for the details.) The program-argument technique is simple, but it is not realistic to demand that all input data be supplied to an application the moment it is started—indeed, most programs use *interactive input*, where the program lets its user submit data while the program executes.

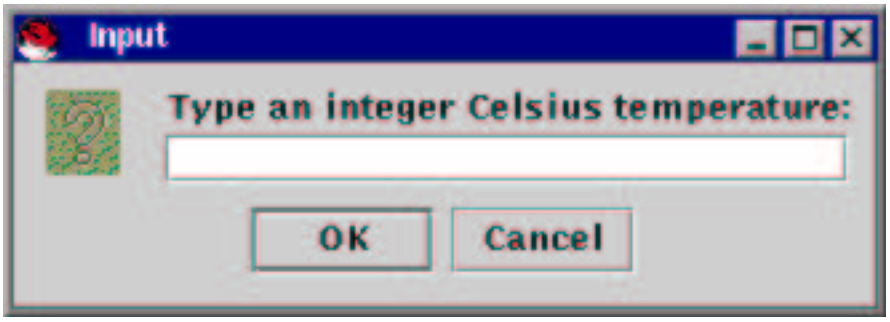
To implement interactive input, an application must use an *input-view* component, as displayed in Figure 1.

The Figure makes several points: First, the application is built from several classes, where the "start-up class" contains the `main` method. Since this class controls what happens next, we call it the *controller*. The controller asks another component to read inputs that the user types; the part of the program that is responsible for reading input is called the *input-view*. The controller uses yet another part for displaying output; this is the *output-view*.

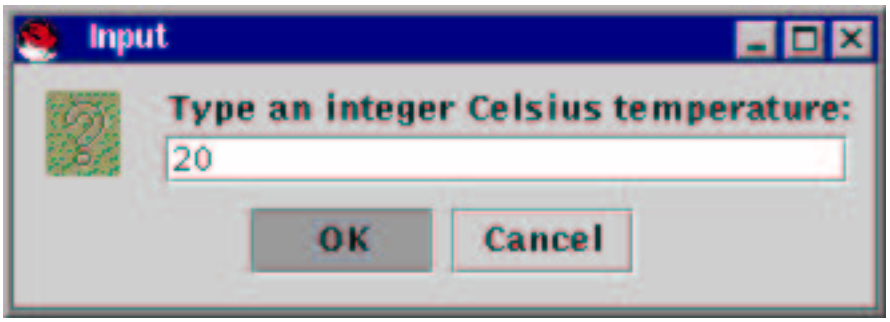
The programs in Chapters 2 and 3 had no input-view, and `System.out` was used as a simple output-view. In this section, we improve the situation by adding an input-view that interacts with the user to receive input.

It is best to see this new concept, the input-view, in an example: Say that we revise the temperature-conversion application so that its user starts it, and the application

displays an *input dialog* that requests the input Celsius temperature:



The user interacts with the input dialog by typing an integer, say `20`, and pressing the *OK* button:



The dialog disappears and the answer appears in the command window:

```
For Celsius degrees 20,
Degrees Fahrenheit = 68.0
```

It is the input-view object that interacts with the user and reads her input. The extensive library of Java packages includes a class that knows how to construct such input dialogs; the class is `JOptionPane`. The crucial steps for using the class to generate an input-reading dialog are

1. At the beginning of the application, add the statement,

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

   so that the Java interpreter is told to search in the `javax.swing` package for `JOptionPane`.

2. Create the dialog on the display by stating

   ```
   String input = JOptionPane.showInputDialog(PROMPT);
   ```

Figure 4.2: Interactive input from a dialog

```
import java.text.*;
import javax.swing.*;
/**  CelsiusToFahrenheit2 converts an input Celsius value to Fahrenheit.
  *    input: the degrees Celsius, an integer read from a dialog
  *    output: the degrees Fahrenheit, a double */
public class CelsiusToFahrenheit2
{ public static void main(String[] args)
  { String input =
      JOptionPane.showInputDialog("Type an integer Celsius temperature:");
    int c = new Integer(input).intValue();  // convert  input  into an int
    double f = ((9.0 / 5.0) * c) + 32;
    DecimalFormat formatter = new DecimalFormat("0.0");
    System.out.println("For Celsius degrees " + c + ",");
    System.out.println("Degrees Fahrenheit = " + formatter.format(f));
  }
}
```

where `PROMPT` is a string that you want displayed within the dialog. (For the above example, the prompt would be `"Type an integer Celsius temperature:"`.) The message, `showInputDialog(PROMPT)`, creates the dialog that accepts the user's input. The text string that the user types into the dialog's field is returned as the result when the user presses the dialog's *OK* button. In the above statement, the result is assigned to variable `input`. (Like program arguments, inputs from dialogs are strings.)

Begin Footnote: Perhaps you noted that, although `JOptionPane` is a *class*, an `showInputDialog` message was sent to it—why did we not create an object from `class JOptionPane`? The answer is elaborated in the next Chapter, where we learn that `showInputDialog` is a `static` method, just like `main` is a static method, and there is no need to create an object before sending a message to a static method. Again, Chapter 5 will present the details. *End Footnote*

Figure 2 shows how the temperature-conversion application is modified to use the dialog as its input view. As noted above, the input received by the dialog is a *string*, so we must convert the string into an integer with a `new Integer` "helper object":

```
int c = new Integer(input).intValue();
```

The application's outputs are computed and are displayed in the command window, but later in this chapter we learn how to display outputs in graphics windows of our own making.

Of course, an application can ask its user for as many inputs as it desires. For example, the change-making application in Figure 3, Chapter 3, might be converted

into an application that asks its user for the dollars input and then the cents input—the first two statements in Figure 3's `main` method are replaced by these, which construct two dialogs:

```
String d =
    JOptionPane.showInputDialog("Type an integer dollars amount:");
int dollars = new Integer(d).intValue();
String c =
    JOptionPane.showInputDialog("Type an integer cents amount:");
int cents = new Integer(c).intValue();
```

One obvious question remains: What happens when the user presses the dialog's *Cancel* button instead of its *OK* button? If you try this, you will see that the program prematurely halts due to an *exception*:

```
Exception in thread "main" java.lang.NumberFormatException: null
        at CelsiusToFahrenheit2.main(CelsiusToFahrenheit2.java:11)
```

`JOptionFrame`'s input dialog is written so that a press of its *Cancel* button causes it to return a "no value" string, called `null`. (This is different from the empty string, `""`—`null` means "no string at all.")

Although `null` is assigned to variable `input`, it is impossible for the `new Integer` helper object to convert it into an integer, and the program halts at this point. Exceptions are studied in more detail later.

Another odd behavior is seen when the application is used correctly and displays the converted temperature: The application does not appear to terminate, even though all the statements in its `main` method have completed. Alas, `JOptionPane` is the culprit—it is still executing, unseen.

Begin Footnote: `JOptionPane` and the other graphics classes in this chapter generate separate *threads of execution*, that is, they are separately executing objects. So, even though the `main` method's "thread" terminates, the "thread" associated with `JOptionPane` is still executing (perhaps "idling" is a better description), hence, the application never terminates.

Multiple threads of execution are valuable for programming animations and applications where multiple computations must proceed in parallel. *End Footnote*

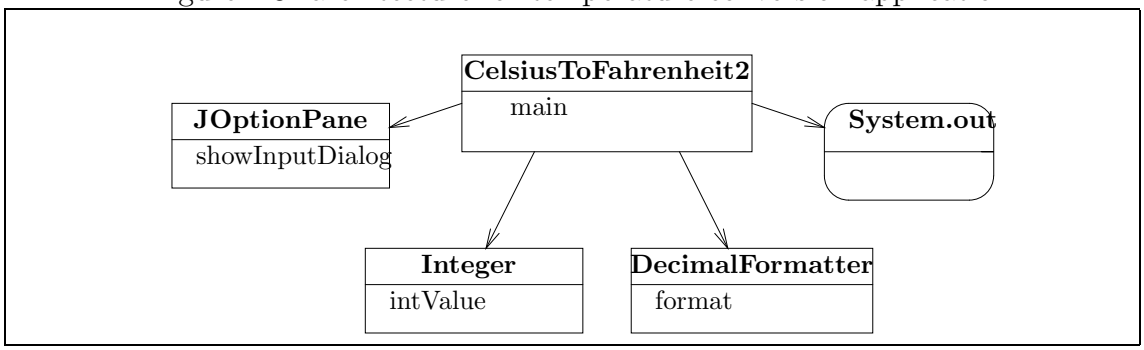
To terminate the application, use the *Stop* or *Stop Program* button on your IDE; if you use the JDK, press the *control* and *c* keys simultaneously in the command window.

Figure 3 displays the class diagram for the application in Figure 2; `JOptionPane` has taken its position as the input view, and the helper classes, `Integer` and `DecimalFormatter`, are included because the controller sends messages to objects created from them.

Exercises

1. Write a sequence of statements that use `JOptionPane` to read interactively a person's name (a string) and the person's age (an integer).

Figure 4.3: architecture for temperature-conversion application

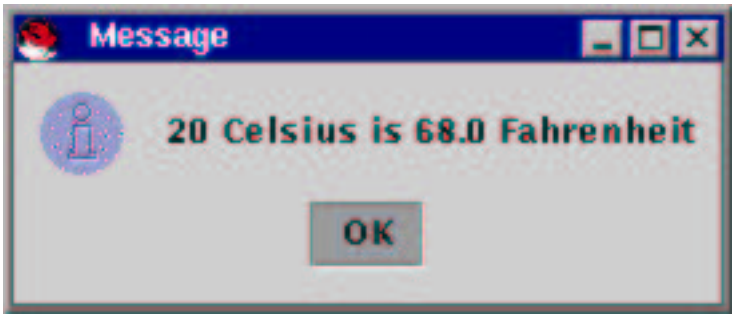


2. Write an application, `class Test`, that reads interactively an integer and prints in the command window the integer's square. Test this program: What happens when you, the user, abuse it by typing your name instead of a number? When you press only the *OK* key for your input? When you refuse to type anything and go to lunch instead?

3. Revise the `MakeChange` program in Figure 3, Chapter 3, so that it uses interactive input.

4.1.1 Dialog Output

If an application's output consists of only one line of text, we might present this line in an *output dialog*, rather than in the command window. Indeed, the output from the previous application might be presented in an output dialog like this:



When the user presses the *OK* button, the dialog disappears.

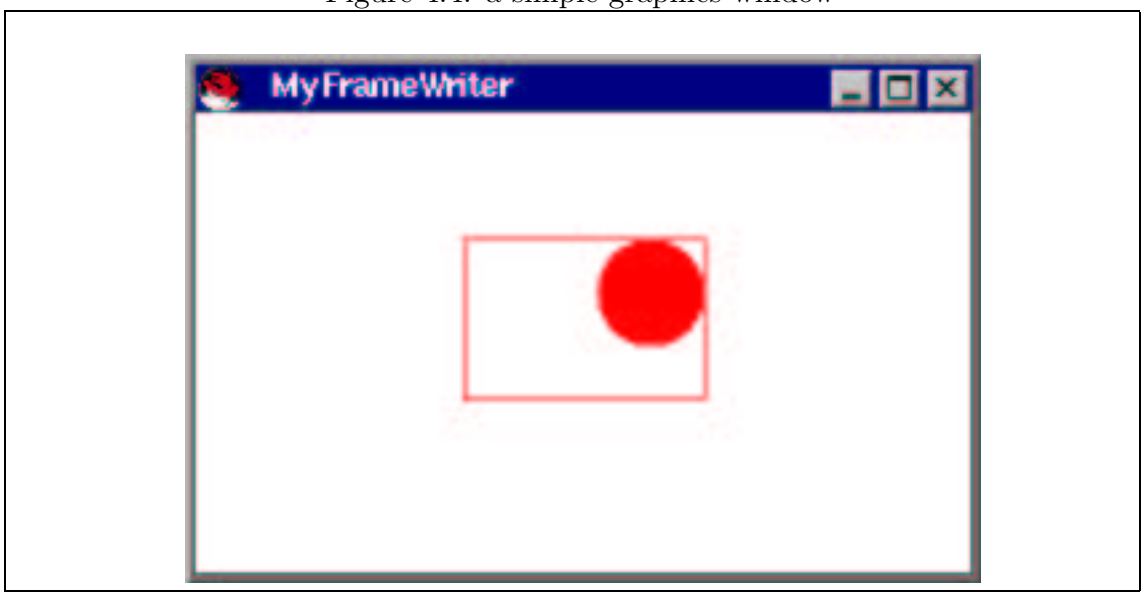
The `JOptionPane` class contains a method, `showMessageDialog`, which constructs and displays such dialogs. The format of messages to `showMessageDialog` goes

```
JOptionPane.showMessageDialog(null, MESSAGE);
```

where `MESSAGE` is the string to be displayed in the dialog. The first argument, `null`, is required for reasons explained in Chapter 10; for now, insert `null` on faith.

We can modify Figure 2 to use the dialog—use this statement,

Figure 4.4: a simple graphics window




```
JOptionPane.showMessageDialog(null, c + " Celsius is "
                                    + formatter.format(f) + " Fahrenheit");
```

as a replacement for the two `System.out.println` statements at the end of the `main` method in Figure 2.

4.2 Graphical Output

We now build output-views that replace `System.out` in our applications. The views we build will be "graphics windows" that display words, shapes, and colors on the display. Graphics windows are objects, and like all objects they must be constructed from classes. In the Java libraries, there already exist classes from which we can construct empty windows. To build an interesting window, we use the prewritten classes in a new way—we extend them by *inheritance*.

A simple graphics window appears in Figure 4. It takes tremendous work to build the window in the Figure from scratch, so we start with the classes of windows available in the `javax.swing` library and extend them to create windows like the one in the Figure. To get started, we study a specific form of graphics window called a *frame* and we learn how to "paint" text and diagrams on a *panel* and insert the panel into its frame.

Figure 4.5: creating an invisible frame

```
import javax.swing.*;
/** FrameTest1  creates a frame */
public class FrameTest1
{ public static void main(String[] args)
  { JFrame sample_frame = new JFrame();
    System.out.println("Where is the frame?");
  }
}
```

Figure 4.6: creating an empty frame

```
import javax.swing.*;
/** FrameTest2  creates and displays an empty frame */
public class FrameTest2
{ public static void main(String[] args)
  { JFrame sample_frame = new JFrame();
    sample_frame.setSize(300, 200);  // tell frame to size itself
    sample_frame.setVisible(true);   // tell frame to make itself visible
    System.out.println("Frame has appeared!");
  }
}
```

4.2.1 Panels and their Frames

In Java, graphics text and drawings are displayed within a frame. Frames are built from `class JFrame`, which lives in the package, `javax.swing`. Any application can use `class JFrame`; see Figure 5.

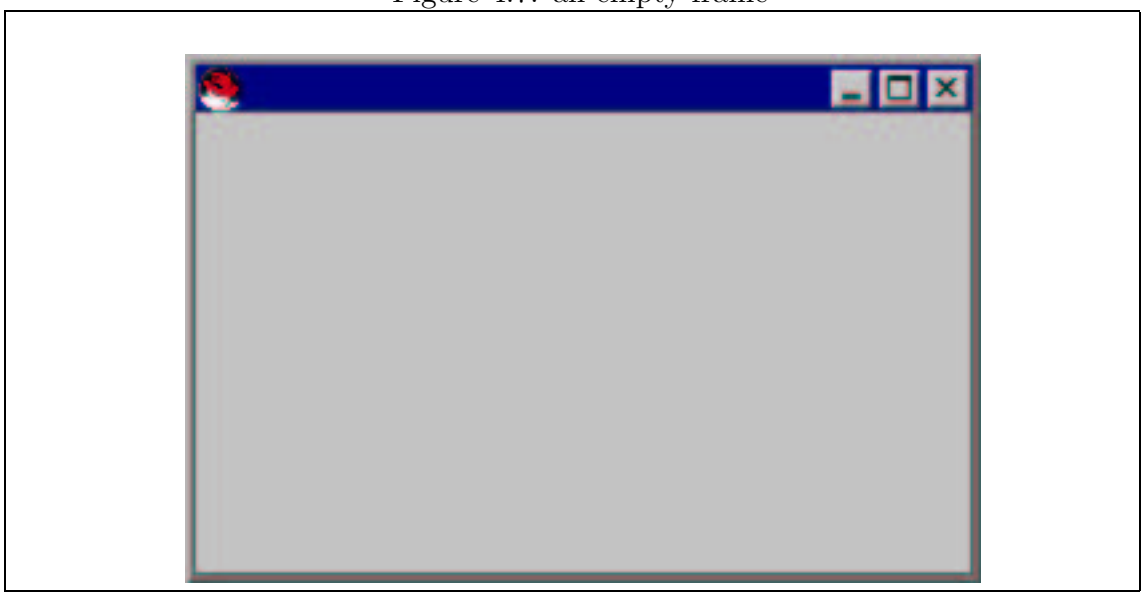
The frame is created by the phrase, `new JFrame()`, and we name the frame, `sample_frame`. As usual, the appropriate `import` statement must be included.

If you execute this example, you will be disappointed—no frame appears, and the application itself does not terminate, even though all of `main`'s statements have successfully executed. (As noted in the previous section, you must terminate the application manually.)

We can repair the situation by sending messages to the frame object, telling it to size itself and show itself on the display. Figure 6 shows how, by sending the messages `setSize` and `setVisible` to the `JFrame` object.

The message, `setSize(300, 200)`, tells `sample_frame` to size itself with width 300 pixels and height 200 pixels. (The unit of measurement, "pixels," is explained later. The sizes, 300 and 200, were chosen because frames in 3-by-2 proportion tend to be

Figure 4.7: an empty frame



visually pleasing.) Then, `setVisible(true)` tells `sample_frame` to make itself truly visible; the result is that the frame in Figure 7 appears on the display.

Alas, the frame in Figure 7 is empty—we have inserted nothing within it to display! To remedy this, we must construct a *panel* object and draw text or shapes onto the panel. The panel is "inserted" into the frame so that we can see it on the display.

4.2.2 Customizing Panels with Inheritance

A *panel* is the Java object on which one draws or "paints." The `javax.swing` library contains a `class JPanel` from which we can construct blank panels. To draw or paint on the panel, we write instructions for painting and attach them to the panel using a technique called *inheritance*. We introduce panel-painting-by-inheritance with a simple example.

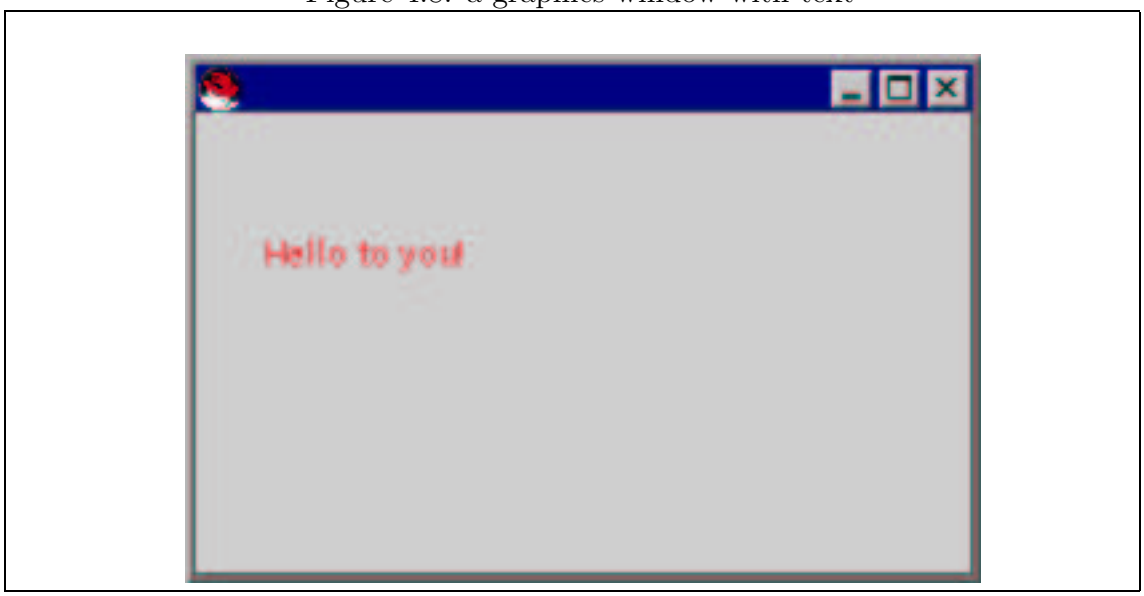
Say that the graphics window in Figure 7 should display the text, `Hello to you!`, in red letters, giving the results in Figure 8. To do this, we take these steps:

1. Starting from `class JPanel`, we write our own customized variant of a panel and name it, say, `class TestPanel`. Within `class TestPanel`, we include the instructions for painting `Hello to you!`, in red letters.

2. We construct an object from `class TestPanel`, we construct a frame object from `class JFrame`, and we insert the panel object into the frame object.

3. We size and show the frame, like we did in Figure 6, and the frame with its panel appears on the display.

Figure 4.8: a graphics window with text



Here is the format we use to write `class TestPanel`:

```
import java.awt.*;
import javax.swing.*;
public class TestPanel extends JPanel
{
  public void paintComponent(Graphics g)
  { ... // instructions that paint text and shapes on the panels's surface
  }
}
```

The crucial, new concepts are as follows:

- `public class TestPanel` asserts that we can construct `TestPanel` objects from the new class; we do it by stating, `new TestPanel()`.

- The phrase, `extends JPanel`, connects `class TestPanel` to `class JPanel`—it asserts that every `TestPanel` object that is constructed will have the contents of a prebuilt, blank `JPanel` object plus whatever extra instructions we have written within `class TestPanel`.

  Here is some terminology: `TestPanel` is called the *subclass*; `JPanel` is the *superclass*.

  When we write `class TestPanel`, *we do not edit or otherwise alter the file where* `class JPanel` *resides.*

Figure 4.9: a panel that displays a string

```
import java.awt.*;
import javax.swing.*;
/** TestPanel creates a panel that displays a string  */
public class TestPanel extends JPanel
{ /** paintComponent  paints the panel
    * @param g - the ‘‘graphics pen’’ that draws the items on the panel */
  public void paintComponent(Graphics g)
  { g.setColor(Color.red);
    g.drawString("Hello to you!", 30, 80);
  }
}
```

- The extra instructions within `class TestPanel` are held in a newly named method, `paintComponent`. Like method `main`, `paintComponent` holds instructions to be executed. Specifically, `paintComponent` must hold the instructions that state how to paint text, shapes, and colors on the panel's surface.

  The `paintComponent` method is started automatically, when the panel appears on the display, and is restarted automatically every time the panel is iconified (minimized/closed into an icon) and deiconified (reopened from an icon into a window) or covered and uncovered on the display.

  `paintComponent`'s header line contains the extra information, `Graphics g`. Every panel has its own "graphics pen" object that does the actual painting on the panel's surface; `g` is the name of the pen. (We study this issue more carefully in the next chapter.)

- Finally, the graphics pen uses classes from an additional package, `java.awt`, so the statement, `import java.awt.*`, must be inserted at the beginning of the class.

Figure 9 shows the completed class, `TestPanel`. Within `paintComponent`, we see the statements,

```
g.setColor(Color.red);
g.drawString("Hello to you!", 30, 80);
```

The first statement sends a message to the graphics pen, `g`, asking it to "fill" itself with red "ink." (Colors are named `Color.red`, `Color.black`, etc. See the section, "Colors for Graphics," at the end of the Chapter.) The second message tells `g` to write the string, `"Hello to you!"` on the surface of the window. The numbers, `30` and `80`, state the position on the window where string drawing should start, namely

Figure 4.10: Constructing a panel and inserting it into a frame

```
import javax.swing.*;
/** FrameTest2  creates and displays an empty frame */
public class FrameTest2
{ public static void main(String[] args)
  { // construct the panel and frame:
    TestPanel sample_panel = new TestPanel();
    JFrame sample_frame = new JFrame();
    // insert the panel into the frame:
    sample_frame.getContentPane().add(sample_panel);
    // finish by sizing and showing the frame with its panel:
    sample_frame.setSize(300, 200);
    sample_frame.setVisible(true);
    System.out.println("Frame has appeared!");
  }
}
```

30 pixels to the right of the window's left border and 80 pixels downwards from the window's top. (The unit of measurement, "pixel," will be defined in the next section.)

Finally, the commentary for the `paintComponent` method contains an extra line of explanation of the graphic pen:

```
/** paintComponent  paints the window
  * @param g - the ‘‘graphics pen’’ that draws the items onto the window */
```

The reason for the idiosyncratic phrase, `@param`, is revealed in the next chapter.

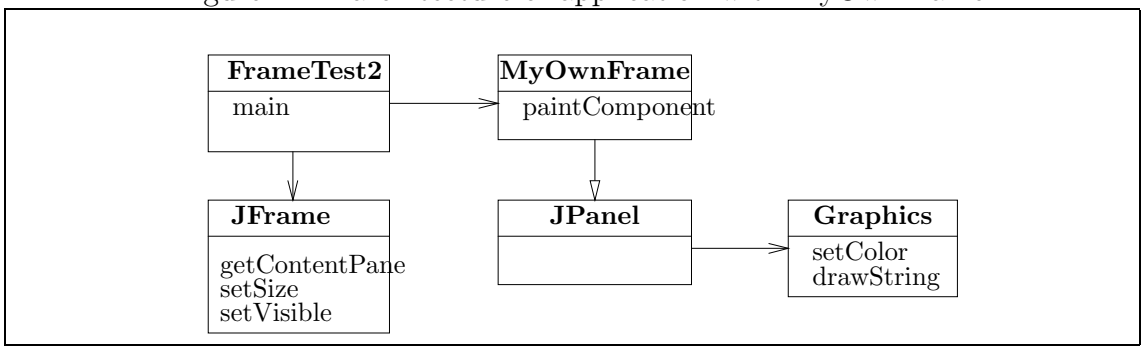
Although we might construct a panel object, by stating, `new TestPanel`, *we cannot execute* `class TestPanel` *by itself.* A panel must be inserted into a frame object, so we write a controller class whose `main` method constructs a panel and a frame, inserts the panel into the frame, and shows the frame. Figure 10 shows us how.

The first statement within the `main` method constructs a panel object from `class TestPanel` and names it `sample_panel`, and the second statement constructs a frame object similarly. To insert `sample_panel` into `sample_frame`, we must use the wordy statement:

```
sample_frame.getContentPane().add(sample_panel);
```

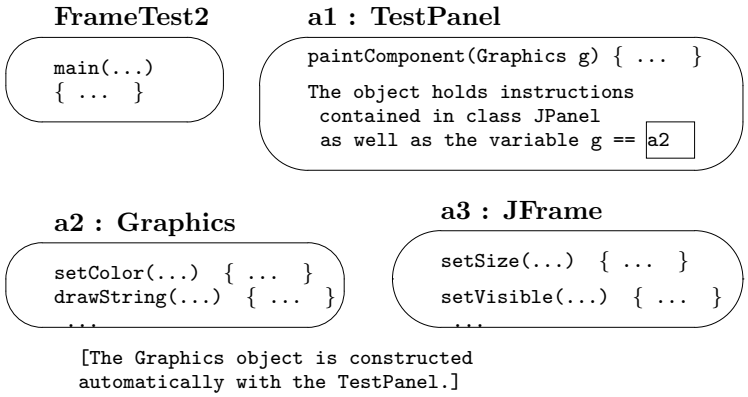
Crudely stated, the phrase, `getContentPane()`, is a message that tells the frame to "open" its center, its "content pane"; then, `add(sample_panel)` tells the content pane to hold the `sample_panel`. A precise explanation of these steps will be given in Chapter 10, but until then, read the above statement as a command for inserting a panel into an empty frame.

Figure 4.11: architecture of application with MyOwnFrame



Once the frame and panel are assembled, we set the size and show the frame as before.

As usual, `class FrameTest2` is placed in a file, `FrameTest2.java`, and `class TestPanel` is placed in the file, `TestPanel.java`. Both files are kept in the same disk folder (directory). When we execute `FrameTest2`, this creates a `FrameTest2` object in primary storage and starts the object's `main` method, which itself creates the `TestPanel` and `JFrame` objects, which appear on the display. After these objects are constructed, the contents of primary storage look like this:



The picture emphasizes that the construction of a `TestPanel` object causes a `Graphics` object (the graphics "pen") to be constructed also. Also, since `TestPanel extends JPanel`, the `TestPanel` object receives the internal variables and methods of an empty `JPanel` object as well as the newly written `paintComponent` method.

Figure 11 shows the architecture of the simple application. The large arrowhead from `MyOwnFrame` to `JFrame` indicates that the former class extends/inherits the latter.

The Figure provides an opportunity to distinguish between the forms of connections between classes: The large arrowhead tells us that a `TestPanel` *is a* `JPanel` but with extra structure. The normal arrowheads tell us that `FrameTest2` *uses a* `JFrame` and a `TestPanel`. The *is-a* and *uses-a* relationships are crucial to understanding the architecture of the application.

We finish this section by presenting the panel that constructs the graphics window in Figure 4; it appears in Figure 12. The controller that uses the panel also appears in the Figure. Before you study the Figure's contents too intently, please read the commentary that immediately follows the Figure.

`Class MyPanel` in Figure 12 contains several new techniques:

- Within the `paintComponent` method, variables like `frame_width`, `frame_heigth`, `left_edge`, `width`, and `height` give descriptive names to the important numeric values used to paint the panel.
- Within `paintComponent`,

  ```
  g.drawRect(left_edge, top, width, height);
  ```

  sends a `drawRect` message to the graphics pen to draw the outline of a rectangle. The `drawRect` method requires four arguments to position and draw the rectangle:

  1. The position of the rectangle's left edge; here, it is `left_edge`.
  2. The position of the rectangle's top edge; here, it is `top`.
  3. The width of the rectangle to be drawn; here, this is `width`.
  4. The height of the rectangle to be drawn; here, this is `height`.

  Thus, the rectangle's top left corner is positioned at the values 105 and 70, and the rectangle is drawn with width 90 pixels and height 60 pixels.
- Similarly,

  ```
  g.fillRect(0, 0, frame_width, frame_height);
  ```

  draws a rectangle, and this time, "fills" its interior with the same color as the rectangle's outline. Of course, `0, 0` represents the upper left corner of the panel.

  Again,

  ```
  g.fillOval(left_edge + width - diameter, top, diameter, diameter);
  ```

  draws and fills an oval. Since a circle is an oval whose height and width are equal, the four arguments to the `fillOval` method are

  1. The position of the oval's left edge, which is computed as `left_edge + width` (the rectangle's right border) minus `diameter` (the circle's diameter).
  2. The position of the oval's top edge, which is `top`.

Figure 4.12: a colorful graphics window

```
import java.awt.*;
import javax.swing.*;
/** MyPanel  creates a colorful panel  */
public class MyPanel extends JPanel
{ /** paintComponent  fills the panelwith the items to be displayed
    * @param g - the ‘‘graphics pen’’ that draws the items */
  public void paintComponent(Graphics g)
  { int frame_width = 300;
    int frame_height = 200;
    g.setColor(Color.white);  // paint the white background:
    g.fillRect(0, 0, frame_width, frame_height);
    g.setColor(Color.red);
    int left_edge = 105;   // the left border where the shapes appear
    int top = 70;          // the top where the shapes appear
    // draw a rectangle:
    int width = 90;
    int height = 60;
    g.drawRect(left_edge, top, width, height);
    // draw a filled circle:
    int diameter = 40;
    g.fillOval(left_edge + width - diameter, top, diameter, diameter);
  }
}


import javax.swing.*;
import java.awt.*;
/** FrameTest3 displays a colorful graphics window */
public class FrameTest3
{ public static void main(String[] args)
  { JFrame my_frame = new JFrame();
    // insert a new panel into the frame:
    my_frame.getContentPane().add(new MyPanel());
    // set the title bar at the top of the frame:
    my_frame.setTitle("MyFrameWriter");
    // an easy way to color the backgrond of the entire window:
    int frame_width = 300;
    int frame_height = 200;
    my_frame.setSize(frame_width, frame_height);
    my_frame.setVisible(true);
    System.out.println("Frame has appeared!");
  }
}
```

3. The width and height of the oval, which are both just `diameter`.

This message draws a circle at the position 155 and 70, of width and height 40 by 40 pixels.

To do a decent job of painting, we must learn more about the format of a window and about the arguments one supplies to methods like `drawRect` and `fillOval`; we study this in the next section.

Within the controller, `FrameTest3`, we see these new techniques:

- One statement both constructs the panel and inserts it into the frame:

```
my_frame.getContentPane().add(new MyPanel());
```

Since the panel is not referenced any more by the controller, there is no need to attach to it a variable name.

- Two more methods of `class JFrame` help construct the window: `setTitle("MyFrameWriter")` places `"MyFrameWriter"` into the window's title bar, and

Exercises

Change `MyPanel` (and `FrameTest3`) as follows:

1. The displayed circle is painted black. (Hint: use `Color.black`.)
2. The size of the panel is 400 by 400 pixels.
3. The background color of the window is yellow. (Hint: use `Color.yellow`.)
4. The oval is painted with size 80 by 40.
5. The rectangle is drawn in the lower right corner of the 300-by-200 window. (Hint: In this case, the upper left corner of the rectangle would be at pixel position 210, 140.)

4.3 Format and Methods for Painting

Every panel and frame, and window in general, has a *width* and a *height*. In Figure 12, the statement, `setSize(300,200)`, sets the width of the window to 300 pixels and its height to 200 pixels. A *pixel* is one of the thousands of "dots" you see on the display screen; it holds one "drop" of color. (For example, to display the letter, "o" on the display, the operating system must color a circle of pixels with black color.)

Within a window, each pixel has a "location," which is specified by the pixel's distance from the window's left border and its distance from the window's *top* (and

Figure 4.13: locating a pixel at coordinates 100,50

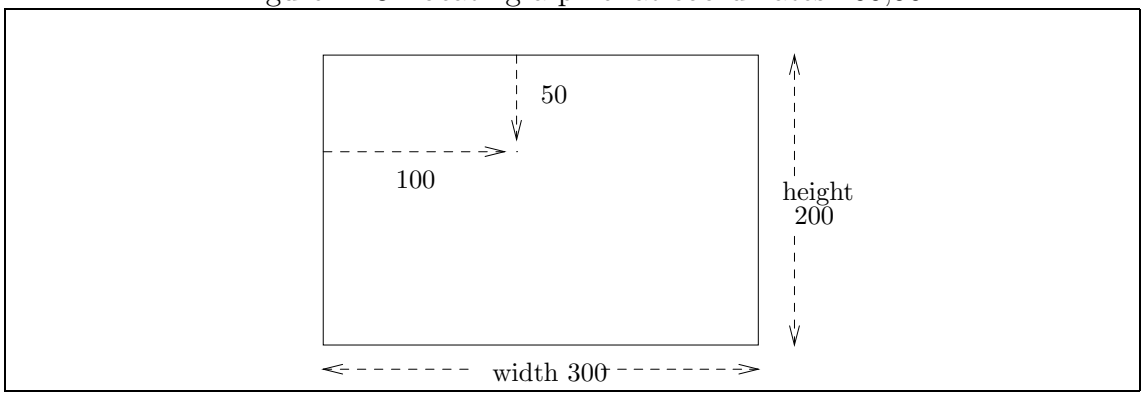
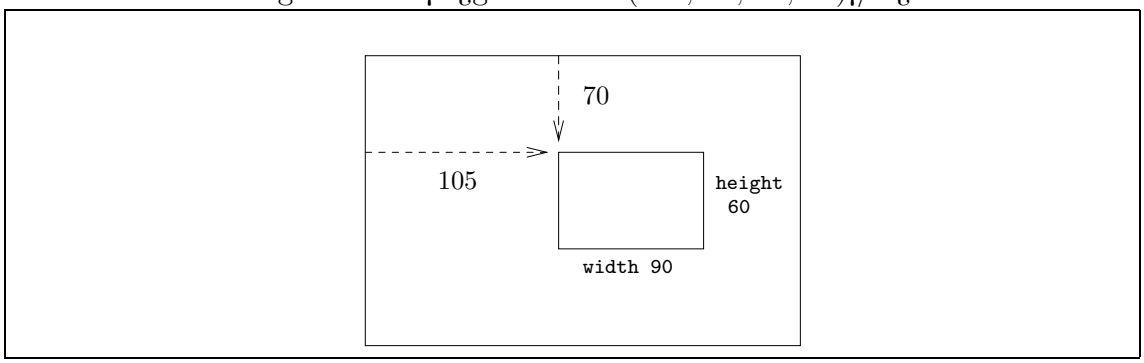


Figure 4.14: ¡tt¿g.drawRect(105, 70, 90, 60)¡/tt¿



not the bottom, contrary to conventional usage). Figure 13 shows how one locates the pixel at position (100, 50) within a window sized 300 by 200.

It is traditional to discuss a pixel's position in terms of its *x,y coordinates*, where x is the horizontal distance from the window's left border, and y is the distance from the window's top border.

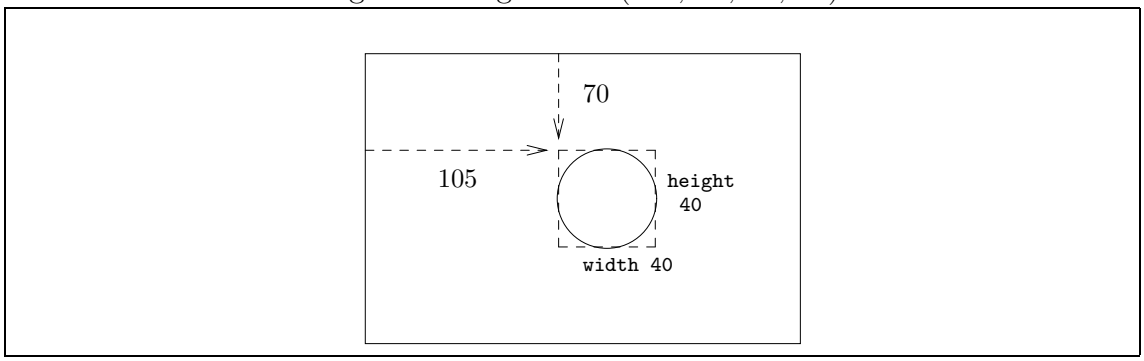
Using pixels, we can give precise directions to a graphics pen: `g.drawRect(105, 70, 90, 60)` tells the graphics pen, `g`, to position the top left corner of a rectangle at coordinates 105,70; the size of the rectangle will be width 90 by height 60. See Figure 14.

To draw an oval, think of it as a as a round shape that must inscribed within an imaginary rectangular box. Therefore, `g.fillOval(105, 90, 40, 40)` tells the graphics pen to position a 40 by 40 oval (a circle of diameter 40) inside an imaginary 40 by 40 box, at coordinates 105, 90. See Figure 15.

The graphics pen has methods to draw lines, text, rectangles, ovals, and arcs of ovals. The methods behave like the two examples in the Figures.

When you paint on a panel, ensure that the coordinates you use indeed fall within the panel's size. For example, drawing an oval at position, 300, 300 within a window

Figure 4.15: g.fillOval(105, 90, 40, 40)



of size 200, 200 will show nothing. Also, assume that a window's top 20 pixels (of its y-axis) are occupied by the title bar—do not paint in this region.

Table 16 lists the methods of `class Graphics` we use for telling a graphics object what to draw. For each method in the table, we see its name followed by a list of the arguments it must receive. (For example, `drawString(String s, int x, int y)` states that any message requesting use of `drawString` must supply three arguments—a string and two integers. For the sake of discussion, the string is named `s` and the integers are named `x` and `y`.)

It might be best to study the details of Table 16 as they are needed in the examples that follow.

4.3.1 Constructor Methods and `this` object

Every panel must be inserted into a frame, and it is tedious to write over and over again the instructions that construct the frame and insert the panel into the frame. It is better to write the panel class so that that the panel *itself* constructs its own frame and inserts itself into the frame. This is done using two new and important techniques: *constructor methods* and *self reference* via `this`.

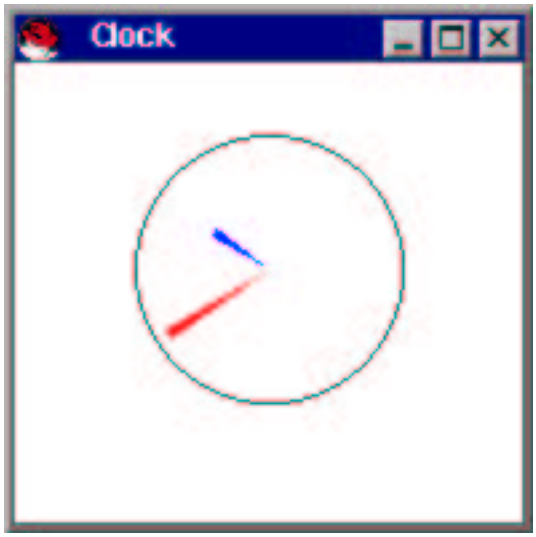
Here is an example that illustrates the new techniques: Say that you want a

Figure 4.16: Graphics pen methods

Methods from `class Graphics`

Name	Description
`setColor(Color c)`	Fills the graphics pen with color, `c`
`drawLine(int x1, int y1, int x2, int y2)`	Draws a line from position `x1, y1` to `x2, y2`
`drawString(String s, int x, int y)`	Displays the text, a String, `s`, starting at position `x, y`, which marks the base of the leftmost letter
`drawRect(int x, int y, int width, int height)`	Draws the outline of a rectangle of size `width` by `height`, positioned so that the upper left corner is positioned at `x, y`
`fillRect(int x, int y, int width, int height)`	Like `drawRect` but fills the rectangle's interior with the color in the pen
`drawOval(int x, int y, int width, int height)`	Draws the outline of an oval of size `width` by `height`, positioned so that the upper left corner of the imaginary "box" that encloses the oval is positioned at `x, y`
`fillOval(int x, int y, int width, int height)`	Like `drawOval` but fills the oval's interior
`drawArc(int x, int y, int width, int height, int start_angle, int thickness)`	Like `drawOval` but draws only part of the oval's outline, namely the part starting at `start_angle` degrees and extending counter-clockwise for `thickness` degrees. Zero degrees is at "3 o'clock" on the oval; the arc extends counter-clockwise for a positive value of `thickness` and clockwise for a a negative value.
`fillArc(int x, int y, int width, int height, int start_angle, int thickness)`	Like `drawArc`, but fills the interior of the partial oval, drawing what looks like a "slice of pie"
`paintImage(Image i, int x, int y, ImageObserver ob)`	Displays the image file (a `jpg` or `gif` image) `i` with its upper left corner at position, `x`, `y`, with "image observer" `ob`. (This method is employed by examples in Chapter 10; it is included here for completeness.)

program that displays the current time as a clock in a window:



As usual, we must write a class that `extends JPanel` whose `paintComponent` method has instructions for drawing the clock's face. But we must also perform the mundane steps of constructing a frame for the panel, inserting the panel into the frame, and displaying the frame. Can we merge these steps with the ones that define and paint the panel? We do so with a *constructor method*.

Every class is allowed to contain a "start up" method, which automatically executes when an object is constructed from the class. The start-up method is called a *constructor method*, or *constructor*, for short. The constructor method typically appears as the first method in a class, and it *must have the same name as the class's name*. For example, if we are writing a panel named `ClockWriter`, then its constructor method might appear as follows:

```
/** ClockWriter  draws a clock in a panel.  */
public class ClockWriter extends JPanel
{ /** ClockWriter is the constructor method:  */
  public ClockWriter()
  { ... // start-up instructions that execute when the
        //  object is first constructed
  }

  /** paintComponent  draws the clock with the correct time.
    * @param g - the graphics pen that does the drawing */
  public void paintComponent(Graphics g)
  { ... }
}
```

Remember that `public ClockWriter()` contains the start-up instructions that execute exactly once, when the `ClockWriter` object is first constructed by means of the phrase,

```
new ClockWriter()
```

Now, we understand that the semantics of the phrase is twofold:

- A `ClockWriter` object is constructed in primary storage;
- A message is sent to the new object to execute the instructions in its constructor method, also named `ClockWriter`.

We wish to write `class ClockWriter` so that it constructs a panel and paints the graphical clock, and we will write the class's constructor method so that it constructs the panel's frame, inserts the panel into the frame, and shows the frame. Figure 17 shows how we do this.

When the instruction, `new ClockWriter()` executes, a `ClockWriter` object is constructed in storage. Then, the constructor method, also named `ClockWriter`, executes:

1. `JFrame clocks_frame = new JFrame()` constructs a frame object in storage.
2. `clocks_frame.getContentPane().add(this)` sends a message to the frame, telling it to add *this* `ClockWriter` panel into it.

   The pronoun, `this`, is used within `ClockWriter`'s constructor method to refer to this newly built `ClockWriter` object whose start-up instructions are executing. When an object refers to itself, it uses `this`, just like you use "I" and "me" when you refer to yourself.
3. The last four instructions in the constructor method tell the frame how to present itself on the display.

Now, whenever you state, `new ClockWriter()`, the panel displays itself in a frame that it constructed itself.

The `paintComponent` method does the hard work of drawing the clock on the panel.

The clock is constructed from a circle and two slender, filled arcs, one arc for the minutes' hand and one for the hours' hand. Based on the current time, in `hours` and `minutes` (e.g., 10:40 is 10 hours and 40 minutes), we use these formulas to calculate the angles of the clock's two hands:

```
minutes_angle = 90 - (minutes * 6)

hours_angle = 90 - (hours * 30)
```

(The rationale is that a clock face contains 360 degrees, so each minute represents 6 degrees and each hour represents 30 degrees of progress on the face. Subtraction from 90 is necessary in both formulas because Java places the 0-degrees position at the 3 o'clock position.)

To fetch the current time, we construct a `GregorianCalendar` object and uses its `get` method to retrieve the current minutes and hours from the computer's clock.

Figure 4.17: class to paint the clock

```
import java.awt.*;
import javax.swing.*;
import java.util.*;
/** ClockWriter  draws a clock in a panel.  */
public class ClockWriter extends JPanel
{ public ClockWriter()
  { int width = 200;  // the width of the clock
    // construct this panel's frame:
    JFrame clocks_frame = new JFrame();
    // and insert _this_ panel into its frame:
    clocks_frame.getContentPane().add(this);
    // show the frame:
    clocks_frame.setTitle("Clock");
    clocks_frame.setSize(width, width);
    clocks_frame.setVisible(true);
  }

  /** paintComponent  draws the clock with the correct time.
    * @param g - the graphics pen that does the drawing */
  public void paintComponent(Graphics g)
  { int width = 200;  // the width of the clock
    g.setColor(Color.white);
    g.fillRect(0, 0, width, width);  // paint the background
    GregorianCalendar time = new GregorianCalendar();
    int minutes_angle = 90 - (time.get(Calendar.MINUTE) * 6);
    int hours_angle = 90 - (time.get(Calendar.HOUR) * 30);
    // draw the clock as a black circle:
    int left_edge = 50;
    int top = 50;
    int diameter = 100;
    g.setColor(Color.black);
    g.drawOval(left_edge, top, diameter, diameter);
    // draw the minutes' hand red, 10 pixels smaller, with a width of 5 degrees:
    g.setColor(Color.red);
    g.fillArc(left_edge + 5, top + 5, diameter - 10, diameter - 10,
              minutes_angle, 5);
    // draw the hours' hand blue, 50 pixels smaller, with a width of -8 degrees:
    g.setColor(Color.blue);
    g.fillArc(left_edge + 25, top + 25, diameter - 50, diameter - 50,
              hours_angle, -8);
  }

  /** The main method assembles the clock in its frame.  The method
    * is inserted here for testing purposes.  */
  public static void main(String[] args)
  { new ClockWriter(); }
}
```

Begin Footnote: The `get` method and its arguments, `Calendar.MINUTE` and `Calendar.HOUR` are explained in Table 22; we do not need the details here. *End Footnote*

Drawing the clock's hands is nontrivial: The minutes' hand must be drawn as a filled arc, slightly smaller than the size of the clock's face. So that we can see it, the arc has a thickness of 5 degrees. A smaller arc is drawn for the hours' hand, and it is attractive to draw it with a thickness of -8 degrees. (See the explanation of `drawArc` in Table 16 for the difference between positive and negative thickness.)

Begin Footnote: Why was the minutes' hand drawn with a positive (counterclockwise) thickness and the hours' hand drawn with a negative (clockwise) thickness? Partly, to show that it can be done! But also because the negative thickness makes the hours' hand more attractive, because this makes the hand "lean" forwards towards the next hour. See Exercise 2 for a better alternative to the negative thickness. *End Footnote.*

Because `class ClockWriter` contains all the instructions for building and displaying the clock, we can write a `main` method that is just this simple:

```
public static void main(String[] args)
{ new ClockWriter(); }
```

There is no need to attach a variable name to the `ClockWriter` object, so it suffices to state merely `new ClockWriter()`.

It seems silly to write a separate controller class that contains just these two lines of text, so we use a standard shortcut—insert the `main` method into `class ClockWriter` as well! At the bottom of Figure 17, you can see the method. You start the `main` method in the usual way, say, by typing at the command line,

```
java ClockWriter
```

The Java interpreter finds the `main` method in `class ClockWriter` and executes its instructions.

Begin Footnote: In the next chapter we learn why the `main` method can be embedded in a class from which we construct objects. *End Footnote*

You should try this experiment: Start `ClockWriter` and read the time. Then, iconify the window (use the window's *iconify/minimize* button), wait 5 minutes, and deiconify ("open") the window—you will see that the clock is repainted with a new time, the time when the window was reopened!

The explanation of this behavior is crucial: *each time a graphics window is iconified (or covered by another window) and deiconified (uncovered), the window is sent a message to execute its* `paintComponent` *method.* For the example in Figure 17, each time the `paintComponent` method executes, it constructs a brand new `GregorianCalendar` object that fetches the current time, and the current time is painted.

Exercises

1. Modify `class ClockWriter` so that its hours' hand is exactly the same length as the minutes' hand but twice as wide.

2. Improve the positioning of the hours' hand in `class ClockWriter` so that it moves incrementally, rather than just on the hour. Use this formula:

   ```
   hours_angle = 90 - ((hours + (minutes/60.0)) * 30)
   ```

3. Make the clock "digital" as well as "analog" by printing the correct time, in digits, at the bottom of the window.

4.4 Objects with State: Field Variables

In Chapter 1, we noted that an object has an "identity," it knows some things, and it is able to do some things. In the Java language, an object's identity is its address in primary storage, and its abilities are its methods.

To give an object knowledge, we equip it with variables that can be shared, read, and updated by the object's methods. Such variables are called *field variables*, or *fields* for short.

Here is a standard use of fields: Perhaps you noticed in the previous two examples (`MyWriter` and `ClockWriter`) that variable names were used to name the width and the height of the panels that were displayed. For example, we saw this coding in the `ClockWriter` example in Figure 17:

```
public class ClockWriter extends JPanel
{ public ClockWriter()
  { int width = 200;  // the width of the clock
        ...
     clocks_frame.setSize(width, width);
        ...
  }

  public void paintComponent(Graphics g)
  { int width = 200;  // the width of the clock
        ...
    g.fillRect(0, 0, width, width);
        ...
  }
}
```

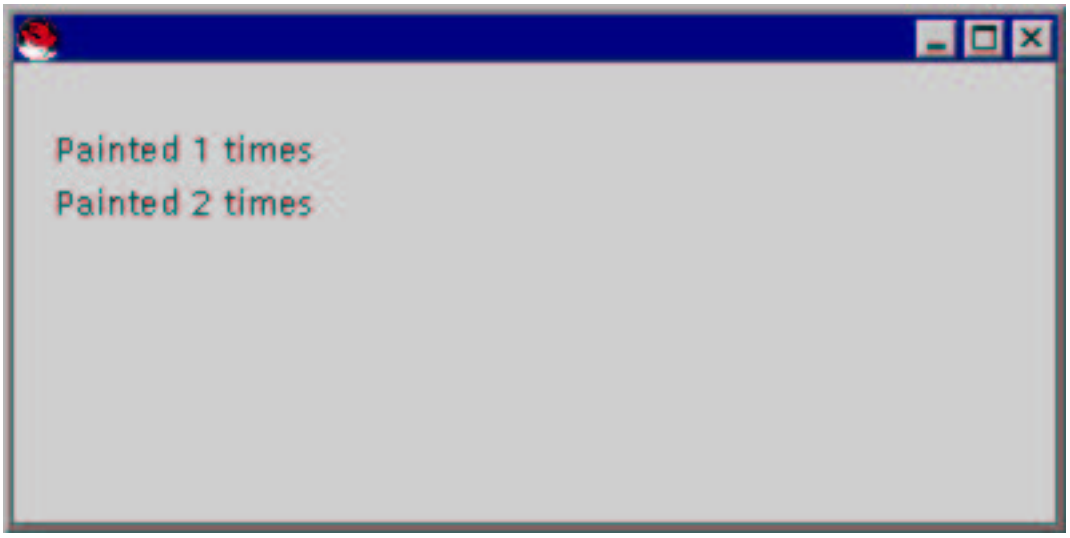
It is unfortunately that we must declare `width` twice, as two local variables; we can make do with one if we change `width` into a field variable and declare it just once, like this:

```
public class ClockWriter extends JPanel
{ private int width = 200;  // the field remembers the panel's width

  public ClockWriter()
  {    ...
     clocks_frame.setSize(width, width);
       ...
  }

  public void paintComponent(Graphics g)
  {    ...
    g.fillRect(0, 0, width, width);
       ...
  }
}
```

Field `width` remembers the panel's width and is shared by both the constructor method and `paintComponent` — it is not "local" or "owned" by either. The field's declaration is prefixed by the keyword, `private`; the keyword means "usable only by the methods in this class (and not by the general public)."

This simple change makes the `class ClockWriter` easier to use and maintain, because a key aspect of the panel's identity — its width — is listed in a single variable that is listed in a key position of the class.

Here is a second example that shows we can change the value of a field variable: Say that we desire a graphics window that "remembers" and prints a count of how many times it has been painted on the display:



A variable is the proper tool for remembering such a count, but the variable cannot be declared inside the object's `paintComponent` method, because each time the

`paintComponent` method is started, the variable will be reinitialized at its starting count. If the variable is declared within the constructor method, then the `paintComponent` method is not allowed to reference it.

Indeed, we have taken for granted a fundamental principle of variables: *a variable declared within a method can be used only by the statements within that method.* Such variables are called *local variables*, because they are "local" to the method in which they are declared. Clearly, a local variable cannot be shared by multiple methods in an object.

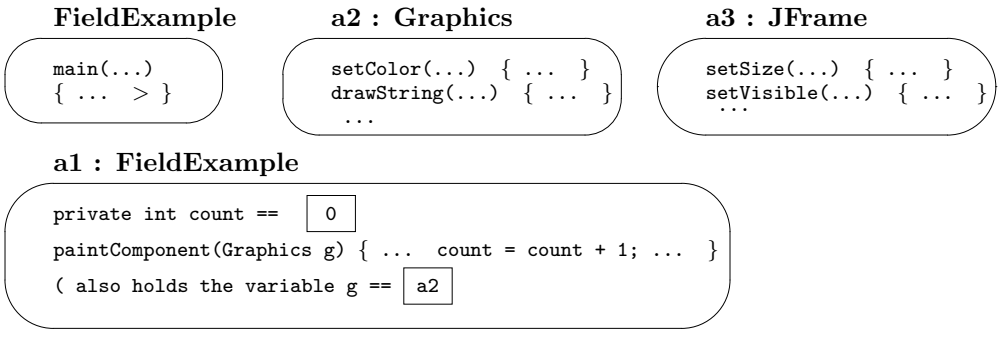
For the example application, we desire a variable that is initialized by the constructor method and is read and updated by the `paintComponent` method. The solution is to declare the variable outside of both the constructor and `paintComponent` methods as a field variable. Figure 18 shows how this is done with a field named `count`.

Field `count`'s declaration is prefixed by the keyword `private`; the keyword means "usable only by the methods in this class (and *not* by the general public)." Because `count` is declared independently from the methods in the class, it is shared by all the methods, meaning that the methods can fetch and change its value.

A field should always appear with an internal comment that explains the field's purpose and restrictions on its range of values and use, if any. (Such restrictions are sometimes called the field's *representation invariant.*) In the Figure, the comment reminds us that `count`'s value should always be nonnegative. This information must be remembered when we write the methods that reference and assign to the field.

The constructor method of Figure 18 sets `count` to zero, to denote that the window has not yet been painted; *it does not redeclare the variable—it merely assigns to it.* Each time a message is sent to `paint`, the `paint` method increments the value in `count` (*it does not redeclare the variable—it merely reads and assigns to it*) and displays the new value.

Here is a picture of how a `FieldExample` object looks in primary storage after its constructor method has finished:



The diagram shows that the field, `count`, is indeed a variable cell whose value can be used by the object's methods (here, `paintComponent`). Also, the graphics pen required by every panel is in fact saved as a field within the panel as well.

Figure 4.18: a field variable

```
import java.awt.*;
import javax.swing.*;
/** FieldExample displays how often a window is painted on the display */
public class FieldExample extends JPanel
{ private int count;  // this field variable holds the count of how
                     // often the window has been painted; for this
                     // reason, its value must always be nonnegative.

  /** FieldExample constructs the window. */
  public FieldExample()
  { count = 0;  // the window has never been painted
    // construct the panel's frame and display it:
    JFrame my_frame = new JFrame();
    my_frame.getContentPane().add(this);
    int height = 200;
    my_frame.setSize((3 * height)/2, height);
    my_frame.setVisible(true);
  }

  /** paintComponent paints the the count of paintings
    * @param g - the graphics pen */
  public void paintComponent(Graphics g)
  { count = count + 1;  // we are painting one more time
    g.setColor(Color.black);
    int margin = 25;
    int line_height = 20;
    int first_line = 40;
    int baseline = first_line + (line_height * count);
    g.drawString("Painted " + count + " times", margin, baseline);
  }

  /** The main method assembles the panel and frame and shows them. */
  public static void main(String[] a)
  { new FieldExample(); }
}
```

Because this example is illustrative and not meant to be part of a serious application, the startup method, `main`, is embedded within `class FieldExample`. This lets us quickly test the class by itself, e.g., `java FieldExample`.

You should compile and execute `class FieldExample`. By iconifying and deiconifying and by covering and uncovering the window, you will learn when and how the `paintComponent` method does its work.

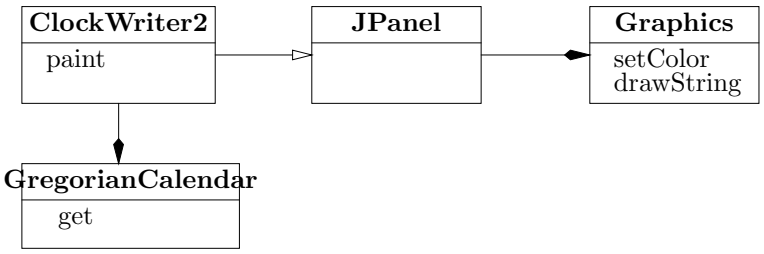
Fields can also hold (the addresses of) objects. For example, we might further rewrite `class ClockWriter` in Figure 17 so that the class retains a `GregorianCalendar` object in a field:

```
import java.awt.*;
import javax.swing.*;
import java.util.*;
/** ClockWriter2  draws a clock in a panel.  */
public class ClockWriter2 extends JPanel
{ private int width = 200;  // the panel's width
  private GregorianCalendar time = new GregorianCalendar(); // the panel's clock

  /** Constructor  ClockWriter2 constructs the properly sized frame */
  public ClockWriter2()
  { ... // the constructor method remains the same
  }

  /** paintComponent  draws the clock with the correct time.
    * @param g - the graphics pen that does the drawing */
  public void paintComponent(Graphics g)
  {  ...
    int minutes_angle = 90 - (time.get(Calendar.MINUTE) * 6);
    int hours_angle = 90 - (time.get(Calendar.HOUR) * 30);
     ... // the remainder of the method stays the same as in Figure 17
  }

  ...
}
```

The field, `time`, is constructed when the object itself is constructed, and it is initialized with the address of a newly constructed `GregorianCalendar` object. When the `paintComponet` method draws the clock, it sends messages to (the object named by) `time` to fetch the time.

When one class, `C1`, constructs an object from class `C2` and retains it within a field, we say that `C1` *has a* (or *owns a*) `C2`. In the above example, `ClockWriter` has a `GregorianCalendar`. The *has-a* relationship is sometimes depicted in a class diagram

by an arrow whose head is a diamond:



Remember these crucial facts about the semantics of fields:

- *The field's cell is created when the object itself is constructed.* If there is an initial value for the field, the value is computed and assigned when the cell is created. The cell lives inside the object, and the field is ready for use when the constructor method starts execution.
- *A field can be initialized when it is declared*, e.g.,

  ```
  private int width = 200;
  ```

  If a field is not initialized, like `count` in Figure 18, the Java interpreter will insert an initial value:

  - Fields that hold numeric values are set to `0`
  - Fields that hold boolean values are set to `false`
  - Fields that hold (addresses of) objects are set to `null` ("no value")

  If a field is not initialized when it is declared, it is best to initialize it within the constructor method.
- *The value in a field's cell is retained even when the object is at rest (not executing one of its methods).* A field acts like an object's "memory" or "internal state," because its value is remembered and can be referenced the next time a message is sent to the object to execute a method.
- *Field names should never be redeclared in a method.* Say that we alter the constructor method in Figure 18 to redeclare `count`:

  ```
  public FieldExample()
  { int count = 0;
    int height = 200;
    setSize(height * 2, height);
    setVisible(true);
  }
  ```

The first statement creates another, distinct, "local" variable that is owned by the constructor method—the `0` is assigned to this local variable and *not* to the field! Unfortunately, the Java compiler allows this dubious double declaration of `count`, and the result is typically a faulty program.

Exercises

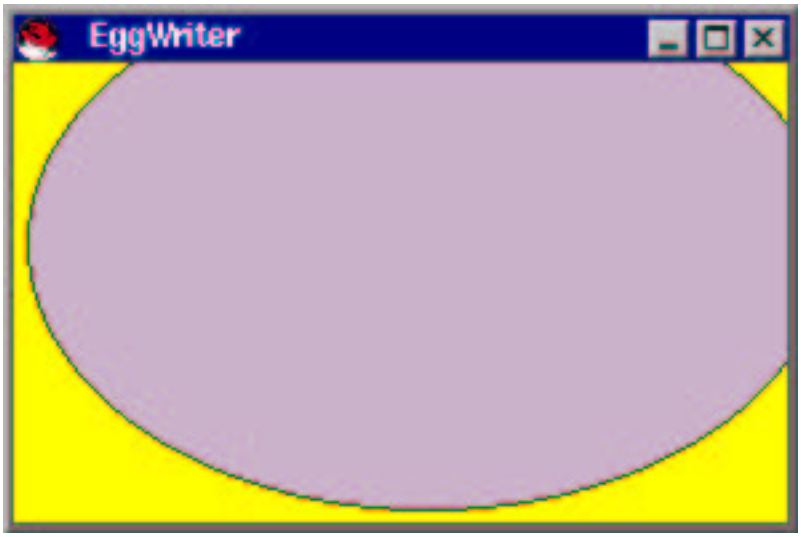
1. Compile and execute this application:

```
public class ShowTwoFieldExamples
{ public static void main(String[] args)
  { FieldExample a = new FieldExample();
    FieldExample b = new FieldExample();
  }
}
```

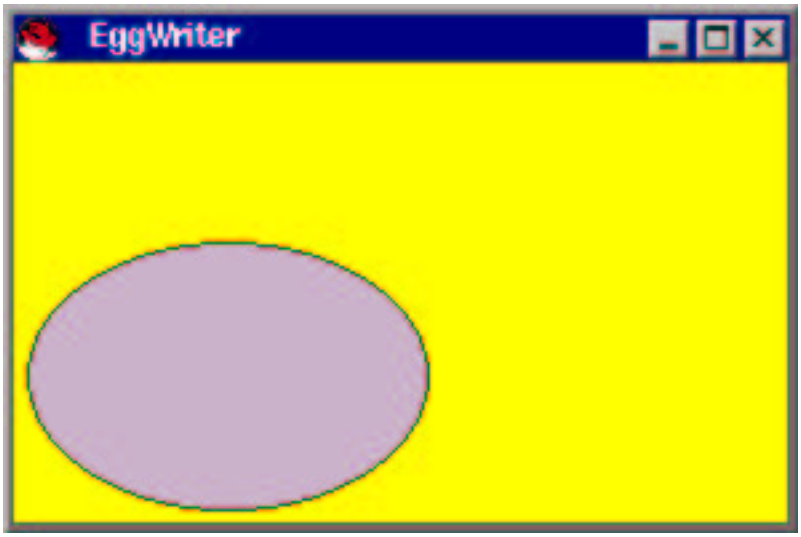
   How many windows does this construct? (Note: Be careful when you answer—try moving the window(s) you see on the display.) When you iconify and deiconify one of the `FieldExample` windows, does this affect the appearance of the other? What conclusion do you draw about the internal states of the respective `FieldExample` objects? Draw a picture of the objects in computer storage created by executing `ShowTwoFieldExamples`.

2. Finish writing `class ClockWriter2` and do an experiment: Start the application, iconify the window it generates on the display, and after five minutes, reopen the window. What time do you see? Constrast this behavior to that of `class ClockWriter` in Figure 17.

3. Here is a humorous example that illustrates field use. Write a `class EggWriter`, that generates a graphics window that displays an egg:



Each time the `EggWriter` window is iconified and deiconified, the egg is repainted at half its former size:

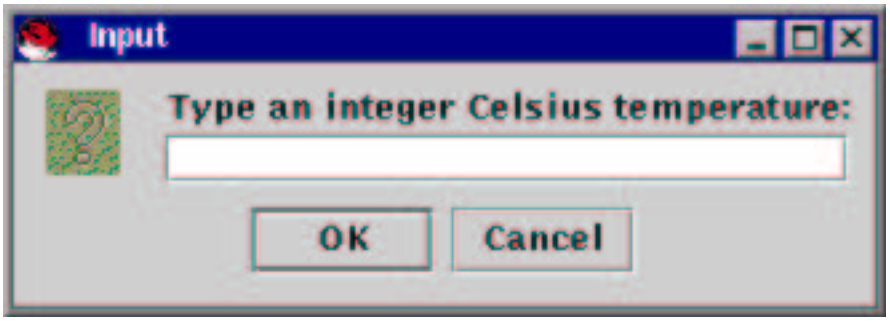


Opening and closing the window enough makes the egg shrink into nothingness. (Hint: Use a field to remember the size of egg to paint; each time the `paint` method redraws the egg, it also halves the value of the field.)

4.4.1 Using Fields to Remember Inputs and Answers

We finish the chapter by combining the techniques we have learned into an application that reads interactive input, computes an answer, and paints the answer into a graphics window. Although the resulting application has a less-than-ideal architecture, we pursue it anyway, because it motivates the materials in the next chapter.

Returning to the Celsius-to-Fahrenheit application in Figure 2, we rewrite it to display its answers in a graphics window. That is, the application generates the dialog seen earlier in the chapter:



and in response to the user's input, a graphics window is constructed that displays

the results:

The application's algorithm is simple:

1. Generate a dialog the reads the input Celsius temperature;
2. Compute the corresponding Fahrenheit temperature;
3. Show a graphics window with the two temperatures.

Because of our limited technical knowledge, it is unclear *how* we design the the graphics window so that it knows which temperatures to print. This problem is simply handled in Chapter 5, but for now we improvise: We write an output-view class, `class CelsiusToFahrenheitWriter`, whose constructor method contains Steps 1 and 2 of the above algorithm. The Celsuis temperature and its Fahrenheit equivalent are saved in fields that the `paintComponent` method uses in Step 3.

The end result is not elegant, but it is effective. Figure 19 shows how we write the constructor method and the `paintComponent` method for `CelsiusToFahrenheitWriter`. Please read the commentary that follows before studying the Figure's contents.

The class's fields are important: `celsius` and `fahrenheit` remember the values of the input and output temperatures, so that the `paint` method can reference them when it is time to print them on the window.

As a matter of style, we added two additional fields, which state basic layout information for the graphics window—if we wish to alter the size or the margins of the window, we revise these field declarations, which are easily spotted because they are located at the beginning of the class and their names are spelled with upper case letters (which is a Java tradition for such "layout" fields).

As promised, the frame's constructor method has packed into it the steps of reading the input and computing the answer; the input and answer temperatures are saved

Figure 4.19: a panel for calculating and displaying temperatures

```
import java.awt.*;
import javax.swing.*;
import java.text.*;
/** CelsiusToFahrenheitWriter creates a panel that displays a
  *  Celsius temperature and its Fahrenheit equivalent.  */
public class CelsiusToFahrenheitWriter extends JPanel
{ int frame_height = 200;
{ private int celsius;        // the input Celsius temperature
  private double fahrenheit;  // a Fahrenheit temperature: its value must
                     //   be the translation of  celsius  into Fahrenheit
  private int LEFT_MARGIN = 20;  // left margin for printing text
  private int LINE_HEIGHT = 20;  // the height of one line of text

 /** Constructor CelsiusToFahrenheitWriter receives the user's input
    *  Celsius temperature, translates it, and displays the output. */
  public CelsiusToFahrenheitWriter()
  { // read the input temperature and compute its translation:
    String input =
      JOptionPane.showInputDialog("Type an integer Celsius temperature:");
    celsius = new Integer(input).intValue();
    fahrenheit = ((9.0 / 5.0) * celsius) + 32;
    // construct the frame and show the answer:
    JFrame f = new JFrame();
    f.getContentPane().add(this);
    f.setTitle("Celsius to Fahrenheit");
    f.setSize((3 * frame_height) / 2, frame_height);
    f.setVisible(true);
  }

  /** paintComponent paints the panel with the two temperatures
    * @param g - the panel's graphics pen */
  public void paintComponent(Graphics g)
  { g.setColor(Color.white);  // paint the background:
    g.fillRect(0, 0, (3 * frame_height) / 2, frame_height);
    g.setColor(Color.red);
    int first_line = LINE_HEIGHT * 4;  // where to print the first line
    g.drawString("For Celsius degrees " + celsius + ",", LEFT_MARGIN,
                 first_line);
    DecimalFormat formatter = new DecimalFormat("0.0");
    g.drawString("Degrees Fahrenheit = " + formatter.format(fahrenheit),
                  LEFT_MARGIN, first_line + LINE_HEIGHT);
  }

  public static void main(String[] args)
  { new CelsiusToFahrenheitWriter(); }
}
```

in `celsius` and `fahrenheit`, respectively. Only then does the constructor method do its usual business of constructing the graphics window.

4.4.2 Scope of Variables and Fields

There are two forms of variables in Java programs: *local variables* and *field variables*. Local variables are declared and used only inside the body of a method; field variables are declared independently of a class's methods and are used by all the methods. The "area of usage" of a variable is called its *scope*.

Here is the technical definition of "scope" for local variable declarations in Java:

> *The scope of a local variable declaration, starts at the declaration itself and extends until the first unmatched right brace,* }.

For example, consider the local variable, `margin`, within the `paintComponent` method in Figure 18:

```
public void paintComponent(Graphics g)
{ count = count + 1;  // we are painting one more time
  g.setColor(Color.black);
  int margin = 25;
  int line_height = 20;
  int first_line = 40;
  int baseline = first_line + (line_height * count);
  g.drawString("Painted " + count + " times", margin, baseline);
}
```

The scope of `margin`'s declaration are these statements:

```
  int line_height = 20;
  int first_line = 40;
  int baseline = first_line + (line_height * count);
  g.drawString("Painted " + count + " times", margin, baseline);
}
```

Thus, `margin` cannot be referenced or changed in the constructor method in Figure 18 nor in any of the statements in `paintComponent` that precede the declaration.

The scope definition for local variables appears to have a problem—consider this example:

```
public static void main(String[] args)   // This method is erroneous!
{ int n = 2;
  System.out.println(n);
  double n = 4.1;         // the scope of  double n  overlaps that of  int n !
  System.out.println(n);  // which  n  is printed?!
}
```

Fortunately, the Java compiler refuses to translate this example, so the problem never arises for an executing program.

On the other hand, it is possible to have two local variables with the same name if their scopes do not overlap. The above example can be inelegantly repaired by inserting an artificial pair of set braces:

```
public static void main(String[] args)
{ { int n = 2;
    System.out.println(n);
  }  // int n's  scope stops here
  double n = 4.1;
  System.out.println(n);   // double n's  value is printed
}
```

There is also a precise definition of scope for a field variable:

> *The scope of a field variable declaration extends throughout the class in which the field is declared, except for those statements where a local variable declaration with the same name as the field already has scope.*

For example, the scope of field `count` in Figure 18 extends through both methods of `class FieldExample`.

Begin Footnote: Unfortunately, the story is a bit more complex than this—the order in which fields are declared matters, e.g.,

```
public class ScopeError
{ private int i = j + 1;
  private int j = 0;
   ...
}
```

generates a complaint from the Java compiler, because the compiler prefers that field `j` be declared before field `i`. *End Footnote.*

Although it is written in bad programming style, here is a contrived example that illustrates scopes of fields:

```
public class Contrived extends JPanel
{ private double d = 3.14;

  public Contrived()
  { System.out.println(s);
    System.out.println(d);
    int d = 2;
    System.out.println(d);
    s = d + s;
    System.out.println(s);
```

```
    setVisible(true);
  }

  private String s = "X" + d;

  public void paintComponent(Graphics g)
  { System.out.println(d + " " + s); }
}
```

This class uses fields, `d` and `s`; when a `Contrived` object is created and its constructor method executes, the constructor prints `X3.14`, then `3.14`, then `2`, and then `2X3.14`. This shows that the scope of `s`'s declaration extends throughout the class, whereas the scope of field `d`'s declaration extends only as far as the declaration of `int d`, *whose scope takes precedence at that point.*

Later, whenever `paintComponent` executes, it prints `3.14 2X3.14`.

The unpleasantness of the example makes clear that it is best *not* to use the same name for both a field variable and a local variable.

4.5 Testing a Program that Uses Input

A programmer quickly learns that merely because a program passes the scrutiny of the Java compiler, it is not the case that the program will always behave as intended. For example, perhaps we were editing the temperature-conversion program in Figure 19 and mistakenly typed `fahrenheit = ((9.0 * 5.0) * celsius) + 32` when we really meant `fahrenheit = ((9.0 / 5.0) * celsius) + 32`. The resulting program will compile and produce wrong answers for its inputs. Therefore, a programmer has the obligation of *testing* a program for possible errors.

Program testing proceeds much like a tool company tests its new tools: A tool is tried on a variety of its intended applications to see if it performs satisfactorily or if it breaks. Of course, a tool cannot be tested forever, but if a tool works properly in testing for some extended period, then the tool company can confidently issue a guarantee with the tools it sells. Such a guarantee typically lists the tolerances and range of activities for which the tool will properly behave and warns the tool's user that performance cannot be guaranteed outside the recommended tolerances and activities.

To test a computer program, one supplies various forms of input data and studies the resulting outputs. The person testing the program might go about this in two ways:

1. The tester studies the statements of the program, and invents inputs that might make the statements malfunction. For example, if a program's statements compute upon dollars-and-cents amounts, like the change-making program does, the

tester tries to trick the program by submitting zeroes or negative inputs. Or, if the program contains a statement like `int x = y / z`, the tester invents inputs that might cause `z` to have a value of 0, which would cause a division-by-zero error. Also, the tester invents enough inputs so that every statement in the program is executed with at least one input. (This helps check the behavior of the `catch` sections of exception handlers, for example.)

This style of testing is sometimes called "glass-box" or "white-box" testing, because the tester looks inside the program and tries to find weaknesses in the program's internal construction.

2. Without looking at the program, the tester pretends to be a typical user and supplies inputs that mimic those that the program will receive in its actual use. The inputs will include common mistakes that a user might make (e.g., typing a fractional number as an input when an integer is required).

   This style of testing is sometimes called "black-box" testing, because the internal structure of the program is not seen, and the tester focusses instead on the program's external, visible behaviors.

In practice, a combination of glass-box and black-box testing is used.

The primary purpose of testing is to identify errors in a program and repair them, but testing also identifies the limits under which a program might be expected to behave properly. For example, a change-making program must behave correctly for positive dollars-and-cents inputs, but if it misbehaves with ridiculous inputs, e.g. a negative dollars value, it is not a disaster—the program can be given to its users with the caveat that correct behavior is guaranteed for only precisely defined, reasonable forms of input data.

Unfortunately, even systematic testing may miss a problematic combination of input data, so testing a program never brings complete confidence in the program's performance. Like the guarantee accompanying the newly purchased tool, the guarantee with a newly purchased program is inevitably limited.

The previous sentence should make you uneasy—perhaps there is a way of validating a program so that it is certain to work correctly for all possible inputs? Indeed, one can employ techniques from mathematical logic to validate a program in this way, but the effort is nontrivial. Nonetheless, some aspects of logical validation of code will be used in subsequent chapters to aid the program development process.

Exercise

Rewrite the change-making application of Figure 3, Chapter 3, so that it requests two integer inputs (a dollars value and a cents value) and makes change. Next, define some "black box" test cases (of input data) for the change-making application in Figure 2, Chapter 4; define some "white box" test cases. Based on your experiments, write a

list of the forms of interactive input will cause the program to operate properly and write a list of the forms of input that might cause unexpected or incorrect answers.

4.6 Summary

The important aspects of this chapter are summarized in the following subsections.

New Constructions

- *class definition by inheritance* (from Figure 9):

```
public class TestPanel extends JPanel
{ ... }
```

- *constructor method* (from Figure 17):

```
/** ClockWriter  draws a clock in a panel.  */
public class ClockWriter extends JPanel
{ public ClockWriter()
  { ... }

  ...
}
```

- *object self-reference* (from Figure 17):

```
public ClockWriter()
{ ...
  JFrame clocks_frame = new JFrame();
  clocks_frame.getContentPane().add(this);
  ...
}
```

New Terminology

- *input view*: the component(s) of an application that receive input data
- *output view*: the component(s) of an application that display output data
- *interactive input*: input data that the user supplies while a program executes—the user "interacts" with the executing program.

- *dialog*: a window that displays a short message and accepts interactive input, say by the user typing text or pressing a button. When the user finishes interaction with the dialog, it disappears.

- `null`: a special Java value that denotes "no value." Attempting arithmetic with `null` or sending a message to it generates an exception (run-time error).

- *constructor method*: a method within a class that is executed when an object is constructed from the class. The constructor method's name must be the same as the name of the class in which it appears.

- *inheritance*: a technique for connecting two classes together so that one is a "customization" or extension of another. If class `C2` inherits/extends class `C1`, then any object constructed from `class C2` will have all the structure defined within `class C1` as well as the structure within `class C2`.

- *subclass* and *superclass*: if `class C2 extends C1`, then `C2` is a subclass of `C1` and `C1` is a superclass of `C2`.

- `this`: a special Java value that denotes "this object."

- *painting (a window)*: drawing text, colors, and shapes on a graphics window; done with the `paintComponent` method.

- *graphics pen*: the object used by a window's `paint` method for painting

- *frame*: Java's name for an application's graphics window.

- *pixel*: one of the "dots" on the display screen; used as a measurement unit for distances within a window.

- *field variable*: a variable declared in a class independently of the class's methods. A field can be referenced and assigned to by all methods in the class.

- *is-a* relationship: if class `C2` extends class `C1`, we say that `C2` is a `C1`.

- *uses-a* relationship: if class `C2` sends messages to objects constructed from class `C1`, we say that `C2` uses a `C1`.

- *has-a* relationship: if class `C2` possesses a field whose value is a `C1`-object that it constructed, we say that `C2` has a `C1`.

Points to Remember

- An application can receive interactive input and display simple output by means of dialogs. In Java, class `JOptionPane` simply generates dialogs for input and output.

- An application can display texts, colors, and shapes in a graphics window. Graphics can be drawn on a panel, constructed from `class JPanel`. A panel must be inserted into a frame, created from `class JFrame`, to be displayed.

  We use *inheritance* to customize `JPanel` so that it displays graphics. The usual customizations are

  - a constructor method, which sets the window's title and size, and
  - a `paintComponent` method, which lists the shapes and colors to be painted on the panel each time it appears on the display.

- Any object, like a graphics window, can have an internal state. The internal state is represented as field variables, which are declared separately from the methods that use them.

New Classes for Later Use

- `JOptionPane`: Found in package `javax.swing`. Helps construct dialogs with its methods,

  - `showInputDialog(PROMPT)`: displays the string, `PROMPT`, and returns the string typed by the user
  - `showMessageDialog(null, MESSAGE)`: displays the string, `MESSAGE`, to the user

- `class JPanel`: Found in package `javax.swing`. Constructs an empty component, but is simply extended by inheritance to display graphics. Table 21.

- `class JFrame`: Found in package `javax.swing`. Constructs empty, zero-sized graphics windows, but it can be sent messages to set its size and display itself. Some of its methods are listed in Table 21.

- `Graphics`: A class found in package `java.awt`. The graphics pen used in a window's `paint` method is in fact an object constructed from class `Graphics`. Some of the class's methods are listed in Table 16.

4.7 Programming Projects

1. For each of the programming projects you completed from the previous chapter, Chapter 3, revise that project to use interactive input. For example, perhaps you built the application that computes the distance an automobile travels based on its initial velocity, V_i, its acceleration, a, and the time, t: $distance = V_i t + (1/2)a(t^2)$ Convert that application into an interactive one: When it starts, the application greets its user as follows:

   ```
   Please type an initial velocity, in miles per hour (a double):
   ```

   When the user enters a number, the application replies,

   ```
   Next, please type the automobile's acceleration (a double):
   ```

   and

   ```
   Finally, please type the elapsed time in hours (a double):
   ```

   The application grabs the inputs, calculates its answer and displays it.

2. Write an application that helps a child learn multiplications. The program asks the child to type two integers and what the child believes is the product of the integers. Then, the program checks the child's calculations and prints the result. The interaction might go as follows:

   ```
   Type an integer:
   (the child types) 13
   Type another:
   (the child types) 11
   Guess the answer:
   (the child types) 143

   The guess was true---the answer is 143.
   ```

3. In the `java.lang` package there is a method named `Math.random`: using `Math.random()` produces a (pseudo-)random double between 0.0 and 1.0. Use this method to modify the multiplication-drill program in the previous exercise so that the program randomly picks the two integers that the child must multiply. For example, the program says:

   ```
   What is 4 times 8?
   ```

   and the child types her guess, say, 33, and the program replies:

```
What is 4 times 8?
33
The guess was incorrect---the answer is 32.
```

The child starts the application by stating the upper bound, `n`, for the integers, and this ensures that the program generates two integers in the range 0..`n`.

4. Write an application that uses interactive input to help a home buyer calculate her expenses for buying and maintaining a house. The program asks the user to type the following input values:

   - the price of the house
   - the down payment on the loan for the house
   - The number of years for repaying the loan.
   - the monthly utility expenses

   In response, the application prints

   - The monthly expense of keeping the house. (Note: this amount does not include the down payment.)
   - The annual expense of keeping the house. (Again, exclude the down payment.)
   - The total expense of keeping the house for the number of years needed to repay the loan. (Include the down payment in this amount.)

   The application does not worry about inflation costs in its calculations.

   Use this formula to compute the monthly payment on the loan for the house: $payment = \frac{\{}{(}1 + i)^y * p * i\}\{(1 + i)^y - 1) * 12\}$ where `y` is the number of years for repaying the loan, and `p` is loan's *principal*, that is, the price of the house minus the down payment.

5. Write an application that uses interactive input to help the user calculate her monthly living expenses. The inputs requested by the program are

   - the amount of money the user has budgeted each month for expenses
   - the monthly rent
   - the weekly groceries bill
   - the annual cost of clothing, furniture, and other "dry goods"
   - weekly transportation costs (e.g., gasoline, parking fees, bus tickets)
   - monthly transportation costs (car payments, car insurance)

- monthly utility costs
- weekly costs for entertainment, including lunches, coffees, etc.

The program produces as its output these numbers:

- the total monthly expenses
- the difference—positive or negative—between the money budgeted for expenses and the actual expenses

Assume that a year has 12.17 months and a month has 30 days, that is, 4.29 weeks.

6. The following projects are meant to give you experience at using methods for painting. Program a graphics window that displays

   (a) 4 concentric circles, drawn in the center of the window

   (b) a transparent, 3-dimensional cube

   (c) three eggs, stacked on top of one another

   (d) a "snowman," that is, three white spheres crowned by a black top hat

   (e) a 4 by 4 red-and-black chessboard

   (f) a "bull's eye," that is, 5 alternating red and yellow concentric circles, on a black background

   (g) the playing card symbols, spade, heart, diamond, and club (Hint: Build a heart from a filled arc topped by two filled ovals; use similar tricks for the other symbols.)

   (h) a Star of David, drawn in gray on a white background

   (i) a silly face in various colors

   (j) a pink house with a red door resting on green grass with a blue sky (Hint: for the house's roof, use a partially filled oval.)

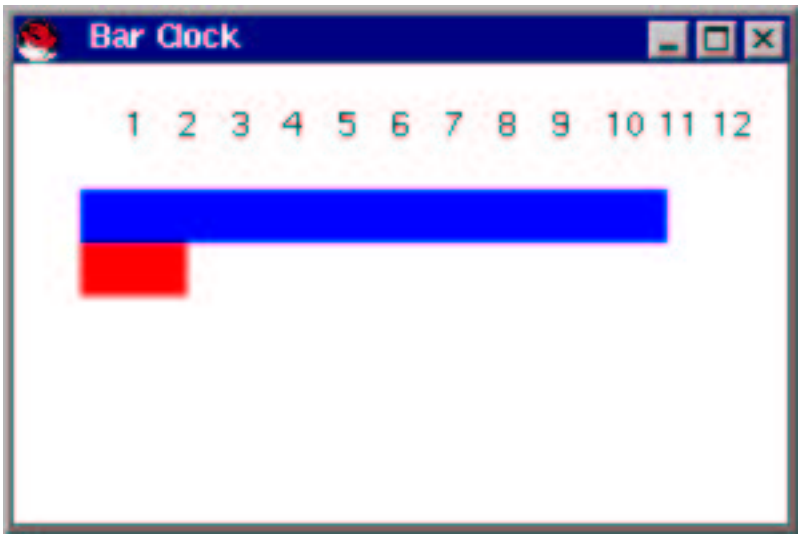
   (k) your initials in blue block letters on a yellow background

7. As noted in Chapter 2, this complex Java statement,

```
try { Thread.sleep(2000); }
catch (InterruptedException e) { }
```

   causes an object to pause for 2 seconds. For any of the graphics windows that you wrote in response to the previous Project exercise, insert copies of this "pause" statement in multiple places within the `paint` method. This will make the parts of your painting appear one part each 2 seconds.

8. Draw a "bar graph" clock that displays the time like this:



   (The time displayed is 11:10.)

9. To draw a line of length `L`, at an angle of `D` degrees, starting at position `x,y`, we must calculate the end point of the line, `x',y'`:

```
      x',y'
       /
      / (angle D)
     /
  x,y . . . . . .
```

   Here are the equations to do this:

```
x' = x + L*(cosine( (D*PI)/180 ) )
y' = y - L*(sine( (D*PI)/180 ) )
```

   (Use `Math.sin` and `Math.cos` calculate sine and cosine, respectively. The value PI is written `Math.PI`.) Use these equations to

   (a) modify `class ClockWriter` in Figure 17 so that the graphical clock is a circle plus one line for the minutes' hand and one line for the hours' hand.

   (b) draw a sun (a circle filled with yellow) that has 6 equally spaced "rays of sunshine" (yellow lines drawn from the center of the circle spaced every 60 degrees).

   (c) improve the display of `class ClockWriter` so that the numerals 1 through 12 are drawn around the border of the clock face in their correct positions. (Note: if you find it too tedious to draw all 12 numerals, draw just 2, 4, and 10.)

Figure 4.20: methods for JFrame

`class JFrame`

Constructor	
`JFrame()`	constructs an empty frame of size 0-by-0 pixels
Method	
`getGraphics()`	Returns as its answer the (address of) this frame's graphics pen
`paint(Graphics g)`	paints on the frame with the graphics pen, `g`
`repaint()`	grabs the graphics pen and sends it in a message to `paint`.
`setLocation(int x, int y)`	sets the frame's location on the display so that its upper left corner appears at location `x`, `y`
`setSize(int width, int height)`	sets the frame's size to `width` by `height`, in pixels
`setTitle(String title)`	sets the frame's title to `title`
`setVisible(boolean yes)`	If `yes` has value `true`, the frame appears on the display; if `yes` has value `false`, the frame disappears

4.8 Beyond the Basics

4.8.1 Partial API for `JFrame`

4.8.2 Methods for `GregorianCalendar`

4.8.3 Colors for Graphics

4.8.4 Applets

The following optional sections develop details of several of the concepts introduced in this chapter. The last section explains how to convert an application into an applet, which is a Java program that can be started within a web browser.

4.8.1 Partial API for `JFrame`

Class `JFrame` possesses a huge collection of methods, and Table 20 summarizes the ones we will use for the next five chapters.

All these methods but `getGraphics` were used in this chapter. As for the latter, it can be used at any time to "grab" a frame's graphics pen. For example, a `main` method can interfere with the appearance of a frame by grabbing the frame's pen and using it:

```
JFrame f = new JFrame();
```

Figure 4.21: class GregorianCalendar

`class GregorianCalendar`

Constructor	
`new GregorianCalendar()`,	creates an object holding the exact time when the object was created
Methods	
`getTime()`	returns the date-and-time held within the object
`get(E)`, where `E` can be any of the arguments listed below	returns the integer value requested by `E`
Arguments for `get` method	
`Calendar.SECOND`	requests the seconds value of the time
`Calendar.MINUTE`	requests the minutes value of the time
`Calendar.HOUR`	requests the hours value (12-hour clock) of the time
`Calendar.HOUR_OF_DAY`	requests the hours value (24-hour clock) of the time
`Calendar.DAY_OF_MONTH`	requests the day of the month of the time
`Calendar.MONTH`	requests the month of the time (Months are numbered 0 to 11.)
`Calendar.YEAR`	requests the current year of the time

Additional methods and arguments can be found in the API for `class GregorianCalendar` in the package `java.util`.

```
Graphics pen = f.getGraphics();
pen.drawString("Hello from main!", 100, 100);
```

This example shows that `getGraphics` returns a result that is a value of type `Graphics`, which is indeed what is indicated in the Table by the phrase, `getGraphics(): Graphics`.

The methods in the Table are part of any object created from a subclass of `JFrame`. Additional methods for `JFrame` are found in `JFrame`'s API in package `javax.swing` and are explained in Chapter 10.

4.8.2 Methods for `GregorianCalendar`

For several chapters we have made good use of `class GregorianCalendar`. A summary of its most useful methods appears in Table 21.

4.8.3 Colors for Graphics

Here are the basic colors you can use to color the background of a window and fill a graphics pen: `black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, yellow, white`. All must be prefixed by `Color.`, e.g., `Color.orange`.

In Java, colors are objects, and it is possible to send a color a message that asks it to create a variant of itself with a slightly different tint, e.g., `Color.orange.brighter()` and `Color.orange.darker()`. Try it: `g.setColor(Color.orange.darker())`. (Of course, you can still use the `Color.orange` object, because it is distinct from the `Color.orange.darker()` object.)

In addition, you can invent your own colors by stating `new Color(r1, g1, b1)`, where all of `r1`, `g1`, and `b1` are numbers between 0 and 255 and are the "red value," "green value," and "blue value," respectively, of the color. For example, `new Color(255, 175, 175)` produces pink and can be used just like `Color.pink`.

You can declare variables to hold (addresses of) colors:

```
private Color BACKGROUND_COLOR = Color.yellow;
```

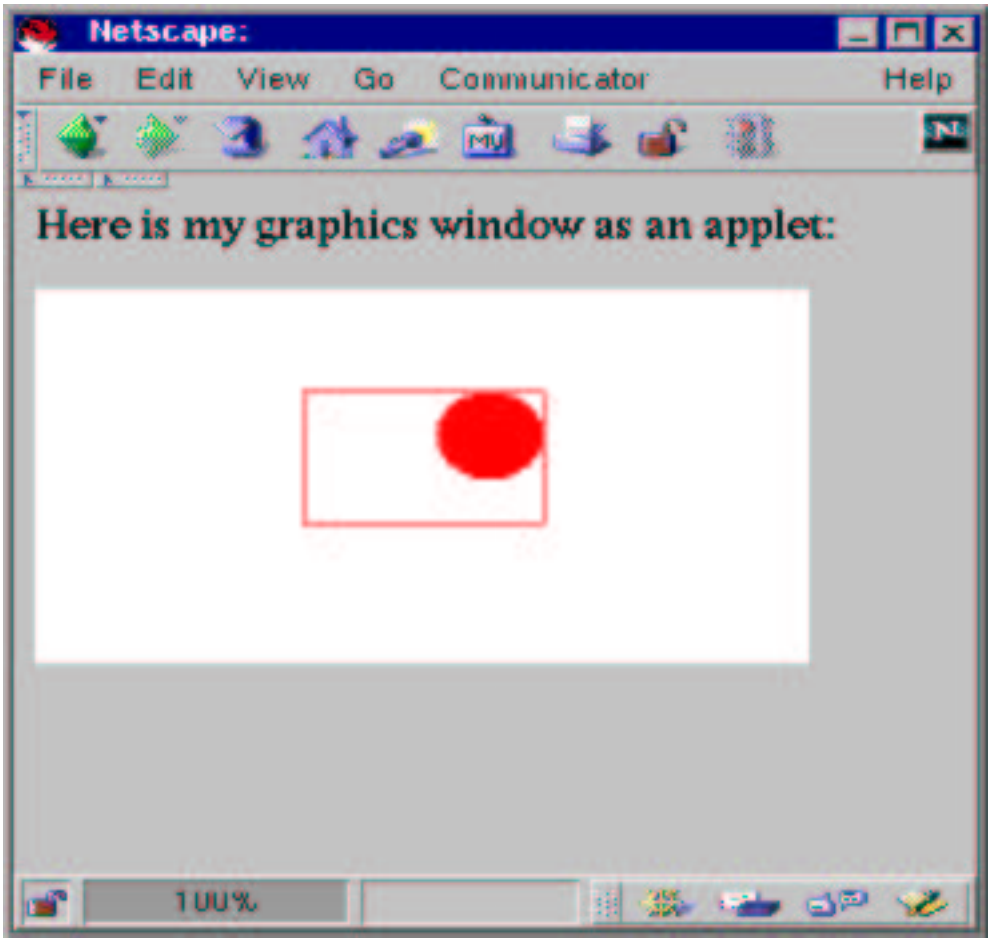
4.8.4 Applets

A Java *applet* is a program whose graphics window appears within a Web page.

Here is some background: You can use a web browser to read files on your computer. Files that contain *hyper-text markup language* (*HTML*) commands will be displayed specially by the browser—you will see multiple columns, multiple type fonts, pictures, and graphics. In particular, the HTML-command, `applet`, starts a Java program (an applet) and displays its graphics window—the browser makes its own copy of the program and executes it.

Thanks to the World-Wide-Web protocol, a web browser can locate HTML-formatted files on other computers in foreign locations. If an HTML-file uses an applet, the file along with the applet are copied into the web browser for display. In this regard, an applet is a "mobile program," moving from browser to browser.

For practice, let's make `class MyPanel` into an applet:



To do this, the "controller" that constructs the panel and displays it is the HTML-file that contains an `applet` command. Figure 22 shows how the HTML-file might appear. The statement, `<applet code = "MyApplet.class" width=300 height=200>`, marks the position where the web browser should insert the graphics window; the width and height of the window are indicated so that the browser can reserve the correct space for the window.

A class that defines a panel ¿ requires several small changes to make it into an applet:

- It `extends JApplet` (rather than `JFrame`).
- *It does not need to be inserted into a frame.* Its "frame" is the web browser itself.
- Its constructor method, if any, is renamed into a method named `init`.
- Its `paintComponent` method is renamed `paint`.

Figure 4.22: The HTML-file ¡tt¿DisplayApplet.html¡/tt¿

```
<head>
<title>My Graphics Window </title>
</head>
<body>
Here is my graphics window as an applet:
<p>
  <applet code = "MyApplet.class" width=300 height=200>
  Comments about the applet go here; these are shown on the web
  page if the applet cannot be displayed. </applet>
</body>
```

Figure 23 shows the applet; compare it to Figure 12.

Remember to compile `MyApplet.java` so that `MyApplet.class` is found in the same folder (directory) as `DisplayApplet.html` When a web browser reads `DisplayApplet.html`, the browser will execute `MyApplet.class`, displaying the applet.

Here is a comment about testing applets: A web browser copies an applet only once, so updating an applet and refreshing the HTML-file that uses it will not show the updates. It is better to test and revise an HTML page and its applet with a testing program called an *appletviewer*. (If you use an IDE, set it to test applets; if you use the JDK, use the command `appletviewer DisplayApplet.html`.) Once the applet works to your satisfaction, only then use the web browser to display it.

Finally, you are warned that there are many different versions of web browsers, and not all web browsers understand all the packages of the Java language. For this reason, applets that tested correctly with an appletviewer may misbehave when started from a web browser, and there is little one can do about it.

Figure 4.23: a simple applet

```
import java.awt.*;
import javax.swing.*;
/** MyApplet displays a window with geometric figures */
public class MyApplet extends JApplet
{
  // An applet has no constructor method---instead, it has an  init  method:
  /** init  constructs the applet for display */
  public void init()
  { }

  // paintComponent  is renamed  paint:
  /** paint fills the window with the items that must be displayed
    * @param g - the ‘‘graphics pen’’ that draws the items onto the window */
  public void paint(Graphics g)
  { int frame_width = 300;
    int frame_height = 200;
    g.setColor(Color.white);  // paint the white background:
    g.fillRect(0, 0, frame_width, frame_height);
    g.setColor(Color.red);
    int left_edge = 105;   // the left border where the shapes appear
    int top = 70;          // the top where the shapes appear
    // draw a rectangle:
    int width = 90;
    int height = 60;
    g.drawRect(left_edge, top, width, height);
    // draw a filled circle:
    int diameter = 40;
    g.fillOval(left_edge + width - diameter, top, diameter, diameter);
  }
}
```