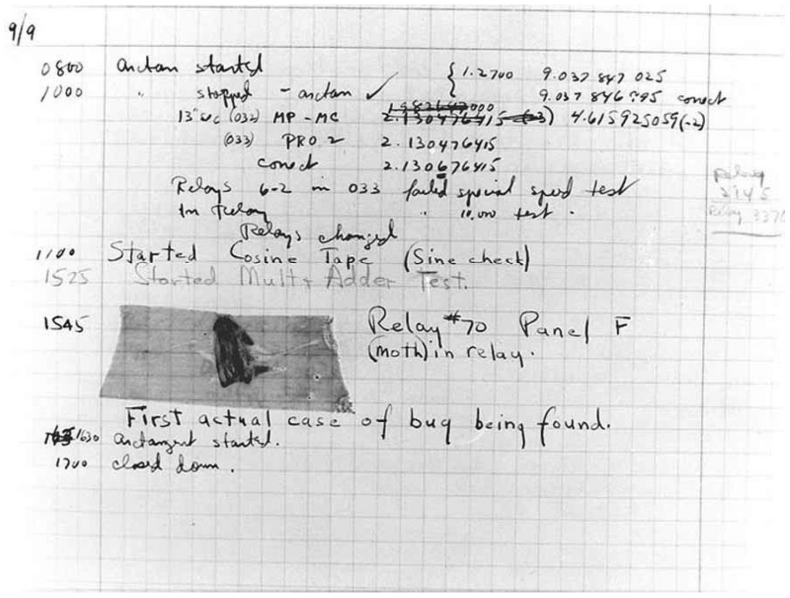
Fortunately, the visual nature of robot programs makes many intent errors easy to find. Debugging intent errors in less visual programs is usually much harder.

1.5.4 A Brief History of Bugs and Debugging

Programming errors of any kind are often called **bugs**. The process of finding and fixing the errors is called **debugging**. The origin of the term is not certain, but it is known that Thomas Edison talked about bugs in electrical circuits in the 1870s. In 1947, an actual bug (a moth) was found in one of the electrical circuits of the Mark II computer at Harvard University, causing errors in its computations (see Figure 1-22). When the moth was found, it was taped into the operator's log book with the notation that it is the "first actual case of a bug being found." Apparently, the term was already in use for non-insect causes of computer malfunctions.



9/9

0800 Arctan started
1000 " stopped - arctan ✓ {1.2700 9.037 847 025
9.037 846 995 conect
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (~~-3~~) 4.615925059(-2)
(033) PRO 2 2.130476415
conect 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 11,000 test .
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Arctangent started.
1700 closed down.

Relay 2145
Relay 3370

(figure 1-22)

A 1947 entry from a log book for the Mark II computer at Harvard University

1.6 GUI: Creating a Window

We have learned a lot of Java programming in the context of `Robot` objects. These concepts include:

- ➤ A class, such as `Robot` or `Thing`, is like a factory for making as many objects as you want. Each class or factory only makes one kind of object.
- ➤ A new object is instantiated with the `new` operator, for instance `Robot mark = new Robot(ny, 0, 2, Direction.WEST);`.

- ➤ All objects belonging to the same class have the same services, but each has its own attribute values that are independent of all other objects.
- ➤ A client can invoke an object's services with the object's name, a dot, and then the name of the desired service.

These concepts are not only for robot programs, but apply to every object-oriented Java program ever written. To illustrate, each chapter of this book includes a section applying the concepts learned using robots to graphical user interfaces, or GUIs (pronounced "gooey"). Applying the concepts to classes supplied with the Java language that have nothing to do with robots shows you how these concepts can be used in many other contexts.

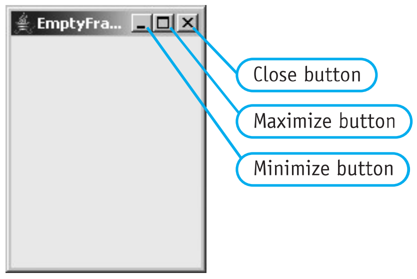
Graphical user interfaces are the part of the program that interacts with the human user. It probably obtains input from the user and displays results to the user. In terms of what the user sees, the GUI consists of the windows, dialog boxes, lists of items to select, and so on.

1.6.1 Displaying a Frame

GUIs add a lot of complexity and development time to a program. Fortunately, Java provides a rich set of resources to help develop interfaces. The beginning point is the window, called a **frame** in Java. The simplest possible frame is shown in Figure 1-23.

(figure 1-23)

Simplest Java frame



Though it is empty, the frame nevertheless has substantial functionality. If the user clicks the close box, the program quits. If the user clicks the minimize box the frame becomes as small as possible, while clicking the maximize box enlarges the frame to take up the entire screen. The user may also adjust the size of the frame by clicking and dragging an edge of the frame.

The program that displays this frame is even simpler than a robot program and is shown in Listing 1-5. Notice the similarity to Listing 1-1, including the following features:

- ➤ Both programs include `import` statements (although the actual packages differ), the declaration of the class at line 3 (although the class name differs), the placement of braces, and the declaration of the special service `main`.

- ➤ Both programs instantiate an object.
- ➤ Both programs invoke the services of an object.

Listing 1-5 *The `EmptyFrame` program displays an empty window*

```
 1 import javax.swing.*;       // use JFrame
 2
 3 public class EmptyFrame
 4 {
 5   public static void main(String[ ] args)
 6   { // declare the object
 7     JFrame frame = new JFrame();
 8
 9     // invoke its services
10     frame.setTitle("EmptyFrame");
11     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12     frame.setLocation(250, 100);
13     frame.setSize(150, 200);
14     frame.setVisible(true);
15   }
16 }
```

FIND THE CODE

ch01/emptyFrame/

PATTERN

Java Program
Object Instantiation

PATTERN

Command Invocation
Sequential Execution

The `EmptyFrame` program differs from the `DeliverParcel` program in the kinds of objects created and the services demanded of them. A `JFrame` object serves as a container for information displayed to the user. By default, however, the frame has no title, is not visible on the screen, has no area for displaying information, and hides the frame when the close box is clicked (leaving us with no good way to stop the program). The services invoked in lines 10-14 override those defaults to provide the functionality we need.

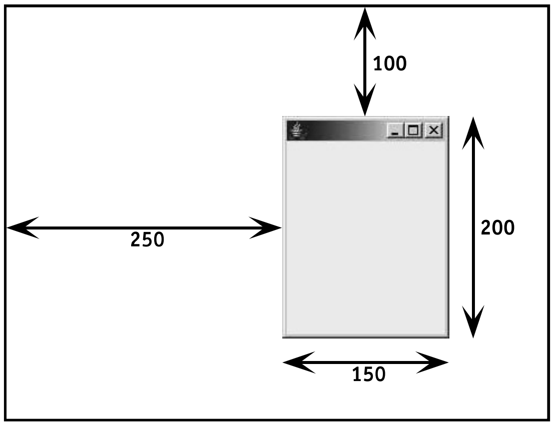
The `setTitle` service causes its argument to be displayed at the top of the frame.

The `setDefaultCloseOperation` service specifies what the frame object should do when the close box is clicked. `JFrame.EXIT_ON_CLOSE` gives a meaningful name to a particular value; `Direction.EAST` serves a similar function for robot programmers.

The `setLocation` service says where the frame should appear. Its first argument is the distance from the left side of the screen to the left side of the frame. The second argument specifies the distance from the top of the screen to the top of the frame. The `setSize` service specifies the size of the frame. The first argument is the width and the second argument is the height. All of the arguments to these two services are given in **pixels**, an abbreviation for **picture elements**, which are the tiny dots on the screen that make up the image. The meaning of these arguments is illustrated in Figure 1-24.

(figure 1-24)

Relationship of the arguments to `setLocation` *and* `setSize` *to the frame's location and size; the outer rectangle represents the computer monitor*



```
JFrame frame = new JFrame();
...
frame.setLocation(250, 100);
frame.setSize(150, 200);
```

LOOKING AHEAD

`true` *and* `false` *are Boolean values used to represent true or false answers, and are covered in detail in Chapter 4.*

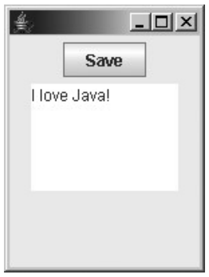
Finally, the `setVisible` service specifies whether the frame is visible on the screen. If the argument is `true`, the frame will be visible, while a value of `false` will hide it.

1.6.2 Adding User Interface Components

A frame with nothing in it is rather boring and useless. We can fix that by setting the **content pane**, the part of a frame designed to display information. We can add to the content pane buttons, textboxes, labels, and other user interface elements familiar to modern computer users. These buttons, labels, and so on are often called **components**. A component is nothing more than an object designed to be part of a graphical user interface.

An early warning, however: The resulting programs may look like they do something useful, but they won't. We are still a long way from writing a graphical user interface that actually accepts input from the user. The following programs emphasize the *graphical* rather than the *interface*.

Figure 1-25 shows a snapshot of a running program that has a button and a text area displayed in a frame. The frame's content pane has been set to hold a `JPanel` object. The `JPanel`, in turn, holds the button and text area. Listing 1-6 shows the source code for the program.



I love Java!

(figure 1-25)

Frame with a content pane containing a button and a text area; the user typed, "I love Java!" while the program was running

Listing 1-6 `FramePlay`, *a program to display a frame containing a button and a text area*

FIND THE CODE

ch01/framePlay/

```
1  import javax.swing.*;        // use JFrame, JPanel, JButton, JTextArea
2
3  public class FramePlay
4  {
5    public static void main(String[ ] args)
6    { // declare the objects to show
7      JFrame frame = new JFrame();
8      JPanel contents = new JPanel();
9      JButton saveButton = new JButton("Save");
10     JTextArea textDisplay = new JTextArea(5, 10);
11
12     // set up the contents
13     contents.add(saveButton);
14     contents.add(textDisplay);
15
16     // set the frame's contents to display the panel
17     frame.setContentPane(contents);
18
19     // set up and show the frame
20     frame.setTitle("FramePlay");
21     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22     frame.setLocation(250, 100);
23     frame.setSize(150, 200);
24     frame.setVisible(true);
25   }
26 }
```

PATTERN

Display a Frame

In lines 7–10, we declare and instantiate several objects. The `JPanel` instance named `contents` is simply a container. It will hold the things we are really interested in, the button and text area. The button and text area are instances of `JButton` and `JTextArea`, respectively. The `JPanel`'s `add` service in lines 13 and 14 adds them to its list of things to display.

When the button is constructed in line 9, `"Save"` is passed to the constructor's parameter. This text appears on the button when it is displayed. When the text area is constructed in line 10, the number of lines of text it should hold and how many characters wide it should be are passed as arguments. Both measures are approximate.

You can learn much more about these classes by browsing the online documentation at *java.sun.com/j2se/1.5.0/docs/api/*.

If you run the `FramePlay` program, you will find that the objects used have quite a bit of built-in functionality. The frame responds to the close, minimize, and maximize boxes, and resizes when you drag an edge. The save button flashes when clicked, and you can type in the textbox. This amount of functionality is remarkable for a 26-line program. As with the robot programs, much of this functionality is due to the programmers who wrote the classes we are using.

KEY IDEA

The ideas you learn with robots—such as classes, objects, and services—apply to all object-oriented programs.

Again, notice the similarity between the `FramePlay` program and all the other programs in this chapter. The concepts we learned with the robot programs truly are general and can be used in all Java programs. In fact, many of the ideas apply to all programs, whether or not they are written in Java. Some ideas, such as modeling, abstraction, and patterns apply to lots of different problems, whether or not they have a computer solution. You are learning a portable set of skills that can be applied in many circumstances.

1.7 Patterns

Many patterns appear in software—problems that appear repeatedly that have the same solution. A number of these have already been identified for you in the margins of this text. Figure 1-26 shows the icon used to identify a pattern. Expert software developers know many patterns and apply an appropriate one to solve the issue at hand, almost without thinking about it. Much of your work, as you learn to program, will involve learning to recognize which software patterns apply to the issue you are facing.

(figure 1-26)

Icon used to identify an example of a pattern

Software patterns began as a way to capture and discuss big ideas, such as how to structure many classes and objects to model a particular kind of problem. In this book we extend the idea of patterns to smaller ideas that may cover only one or two lines of code. As beginning programmers, our attention will be focused primarily on these elementary patterns.