Java Outside In

ETHAN D. BOLKER

BILL CAMPBELL

University of Massachusetts, Boston

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published 2003

Printed in the United States of America

Typefaces ITC Century Book 10/12 pt., Gill Sans, Courier *System* LaTeX 2_ε [TB]

A catalog record for this book is available from the British Library.

Library of Congress Cataloging in Publication Data

Bolker, Ethan D.
Java outside in / Ethan Bolker, Bill Campbell.
p. cm.
Includes bibliographical references and index.
ISBN 0-521-81198-8 – ISBN 0-521-01087-X (pb.)
1. Java (Computer program language) I. Campbell, Bill, 1950– II. Title.
QA76.73.J38 B65 2003
005.13′3–dc21 2002073692

ISBN 0 521 81198 8 hardback
ISBN 0 521 01087 X paperback

Contents

Preface *page* vii

1 Computing with Objects **1**

2 First Things Second **24**

3 Classes and Objects **47**

4 Collections **85**

5 Inheritance **116**

6 Juno **142**

7 When Bad Things Happen to Good Programs **160**

8 Strings **186**

9 Files, Streams, and Persistence **209**

10 Graphical User Interfaces **228**

Glossary **257**

Examples **281**

Index 315

Chapter 1

Computing with Objects

Computers are at work in surprisingly many everyday (and not-so-everyday) places. They control watches, home appliances, cars, airplanes, multinational banking systems, underwater probes, and orbital telescopes. We can use our home computers for everything from word processing, surfing the Internet for business or pleasure, budgeting, and electronic mail to virtual stargazing, language learning, and intergalactic war games. What is remarkable is that all these computers are essentially alike. A computer is a general-purpose machine that can be ***programmed*** to behave as if it were a machine built for a special purpose – a machine for controlling the mixture of gasoline and oxygen in an automobile engine or one for controlling a set of traffic lights. Moreover, the program can be changed without rebuilding the computer, which is why we can use our home computers for so many different tasks. Each ***application*** is a separate program that runs on our computer.

A computer program is a set of instructions (the ***software***) that tells a general-purpose computer (the ***hardware***) how to behave like some special-purpose machine. A computer program is written in a ***programming language***. There are many programming languages; this text uses ***Java***, a general-purpose language based on the modern, object-oriented style of program design that allows one to write programs that focus more on the problem the program must solve than on the computer on which the program will run.

You will learn the craft of programming the way a prospective writer learns the craft of writing. Writers learn by reading. You'll read lots of programs. Writers learn by writing. You'll write many programs.

In this first chapter, we introduce object-oriented programming by studying two applications. The first makes the computer act as if it were an automatic teller machine connected to a bank. The second makes the computer act as if it were a traffic light at an intersection. These programs are longer than those you customarily see in the first chapter of an introductory text. Read them in order to capture a sense of what a Java program looks like. Don't try to understand all the details.

Simulating a Simple Banking System

How does our automatic teller machine (ATM) work? This question is really two different important questions:

- How does the ATM behave? More precisely: How do you interact with the computer when it pretends to be an ATM and you pretend to be a customer? What is the ATM's ***user interface***?

- How does the program we have written cause the computer to behave like an ATM? What is the ATM's ***implementation***?

The Banking System's Behavior

In order to understand the program, we think first about the user interface of the real ATM[1] we are simulating. We insert our bank card into a slot. The ATM reads the card to determine which bank account we want to access. Then it asks for our personal identification number (PIN). When we've entered it correctly, the ATM knows we're entitled to access that account. Then it waits for us to tell it what transactions we want to perform. When we tell it we're done, it returns our card and waits for the next customer.

To keep our simulation simple, our bank will have just two bank accounts. The ATM won't read cards, ask for PINs, or dispense real money. And it will manage its fake money only in whole dollar amounts. We communicate with our ATM by typing messages at the computer keyboard and reading what the computer displays on the screen.

When we start the program, we are telling our computer to behave as if it were an ATM. The computer responds by displaying a welcome message and a prompt asking us to enter an account number. Typing the account number identifies the account we wish to work with – on a real ATM, inserting a bank card does that.

```
Welcome to Engulf and Devour
Account number (1 or 2), 0 to shut down: 1
```

In this example and throughout this book, we use **bold monospaced font** to identify what the computer types and `monospaced font` for what we enter at the keyboard. Remember to press the Return (or Enter) key on your keyboard at the end of each line; until you do, the program will not know that you have typed anything.

Once we choose an account to work with (in this case, account 1), the program offers a list of available transactions and then repeatedly prompts for transactions to process. The program carries out each transaction request, prompting for additional information when necessary. For example, while pretending to be the first customer we might interact with the ATM as follows:

```
Transactions: exit, help, deposit, withdraw, balance
transaction: balance [followed by Return or Enter!]
200
transaction: deposit
amount: 40
transaction: balance
240
```

[1] "Cashpoint" in Canada and the United Kingdom.

```
transaction: withdraw
amount: 25
transaction: balance
215
transaction: help
Transactions: exit, help, deposit, withdraw, balance
transaction: exit
```

When we're finished with an account, we type `exit` as a final transaction. The program ends the transaction cycle for the current account, welcomes the next customer, and prompts for an account number. This in turn begins another transaction cycle for the next account selected:

```
Account number (1 or 2), 0 to shut down: 2
transaction: balance
200
transaction: exit
```

If the first customer now revisits account 1, she will find that her balance is 215.

To shut down the program,[2] we type `0` for an account number:

```
Account number (1 or 2), 0 to shut down: 0
Goodbye from Engulf and Devour
```

Of course we wouldn't allow a customer to shut down a real ATM. Only an employee of the bank would have that authority. But our simulation requires some way to tell our ATM we are done. Entering a zero for an account number is as good a way as any for now.

We will explain soon how you start this ATM program, so that you can duplicate the foregoing dialog rather than just read about it. If you want to do that before reading more about the program, read the "Compile/Run" section of this chapter now.

An Object Model for the Banking System

Now that we understand our application's user interface, we can study its implementation: the program that instructs our computer to behave like an ATM. We designed the implementation by thinking about the objects in the real world that the program must simulate.

Objects are constructs in programming languages that represent things in the real world. In general, anything that we can refer to with a noun, we can represent as an

[2] It's always wise to think before you start work about how you will stop. The brake pedal is more important than the accelerator. In an emergency, you can stop a running Java application by typing Ctrl+C (hold down the Control key as if it were a Shift key and type "C"). What's the best way to shut down your computer?

object in our program. This particular program uses four objects:

- Two **`BankAccount`** objects, one each for each of the two bank accounts being managed.
- One **`Bank`** object representing the bank that maintains the accounts.
- One **`Terminal`** object representing the ATM we use to communicate with the bank.

We begin by thinking about how a **`BankAccount`** object should behave. A real bank account's job is to keep track of how its balance changes as withdrawals and deposits are made. Figure 1-1 shows how we might draw a picture of a **`BankAccount`** object having a balance of 200.

In general, we refer to properties of an object that may change over time as its internal ***state***. Each individual property is stored in a ***field***, a place within the object that we can refer to by name. (These are words you will come to understand as you learn to program in Java. Don't take the time now to memorize the definitions.) The figure shows that each **`BankAccount`** object has a field we chose to call **`balance`** whose value at the moment happens to be 200. A more realistic model of a bank account would have more fields – for the user's name, address and PIN, and other information – and would maintain a balance in pennies.

Objects have ***behavior*** as well as state: They can do things as well as remember the values of their fields. To make an object behave, we send it a ***message***; when the object receives the message, it takes some action. It might ***return*** some value based on its state, or it might change its state.

For example, Figure 1-2 shows a **`BankAccount`** object receiving a **`getBalance`** message and responding by returning the current balance, 200. (Don't worry now about who is sending the message, or what happens to the value 200 that's returned. We will address those questions in time.)

We can change the current balance by sending the **`BankAccount`** a **`deposit`** message, together with an ***argument*** indicating how much is to be deposited. Figure 1-3 illustrates depositing 40. In this case, the **`BankAccount`** object responds to the message by changing the value of its **`balance`** field from 200 to 240. No value is returned, but the object's internal state has changed.

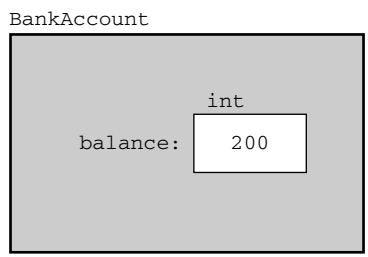

Figure 1-1: A **`BankAccount`** object

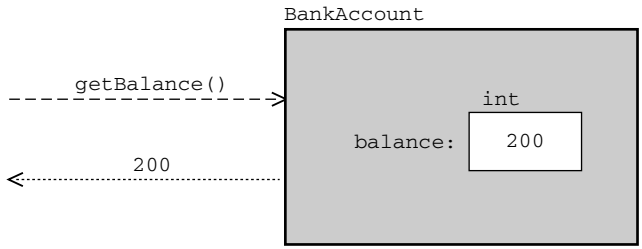

Figure 1-2: Sending a **`getBalance`** message

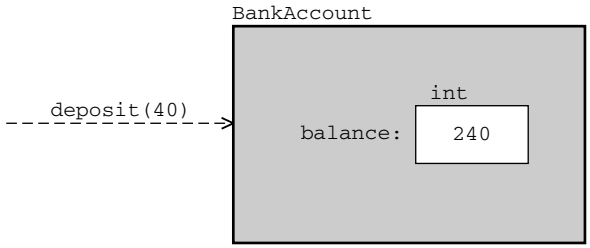

Figure 1-3: Sending a **`deposit`** message

Figure 1-4 depicts the **`Bank`** object in our program. A **`Bank`** has four fields:

- Field **`bankName`** stores the name of the bank – in this case, "Engulf and Devour".[3]
- The values of fields **`account1`** and **`account2`** are the two **`BankAccount`** objects that the bank maintains.
- Field **`atm`** stores a **`Terminal`** object.[4]

In Figure 1-4 the values of the bank's fields appear at the ends of arrows rather than inside the bank itself. That's because those values are other objects in the program.

The Simple Banking Program in Java

Now that we have designed the object structure for our banking system, we will see how to express that structure in Java. In Java, as in all object-oriented languages, each particular object is an ***instance*** of a ***class***. The two **`BankAccount`** objects are instances of class **`BankAccount`**, the one **`Bank`** object is an instance of class **`Bank`**, and the one **`Terminal`** object is an instance of class **`Terminal`**. The string **`"Engulf`**

[3] The name comes from the Mel Brooks film *Silent Movie*.
[4] In a more extensive example, where we had an ATM connected to a network of many banks, we might want to put the **`Terminal`** object elsewhere, because it would not belong to a single **`Bank`**.

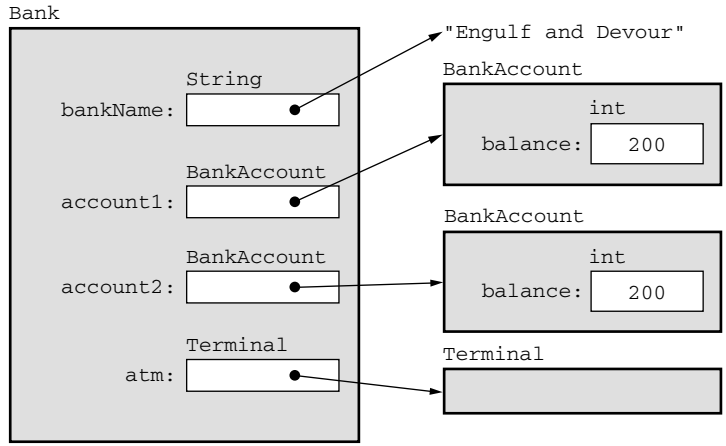

Figure 1-4: A **Bank** object and its fields

and Devour" is an instance of class **String**. The definition of the **BankAccount** class specifies the fields and the behavior of the objects that are instances of the class. This is just as it should be, because all instances of the class share the same properties. They have the same fields, though those fields may have different values in different instances – every **BankAccount** has a balance, but of course those **balance** fields have different values in different accounts.

Listing 1-1 is the Java ***class definition*** for the **BankAccount** class. In Java, each class definition is stored in a computer file whose name is the name of the class being defined with a **.java** suffix (called an ***extension***) appended,[5] so that listing is just the contents of the file **BankAccount.java**.[6] Such a file is often referred to as a ***source file;*** the Java code in it is called ***source code***.

Listing 1-1: BankAccount.java

```
1  // joi/1/bank/BankAccount.java
2  //
3  //
4  // Copyright 2003 Bill Campbell and Ethan Bolker
5
6  /**
```

[5] Usually. Later we will see a few situations in which a Java file contains the definition of several classes.

[6] In this and the other program listings in this book, the line numbers at the left are not part of the program; they are there so that we can easily refer to specific lines of code in our discussions. Listing 1-1 is displayed in full right here. Future listings will be longer. In order not to interrupt the flow of the narrative in the text, you will find all of them on the CD-ROM supplied with this text, and on the Web at `www.cs.umb.edu/joi` so that you can look at them online while reading the text.

```
 7   * A BankAccount object has a private field to keep
 8   * track of this account's current balance, and public
 9   * methods to return and change the balance.
10   *
11   * @see Bank
12   * @version 1
13   */
14
15  public class BankAccount
16  {
17      private int balance; // work only in whole dollars
18
19      /**
20        * A constructor for creating a new bank account.
21        *
22        * @param initialBalance the opening balance.
23        */
24
25      public BankAccount( int initialBalance )
26      {
27           this.deposit( initialBalance );
28      }
29
30      /**
31        * Withdraw the amount requested.
32        *
33        * @param amount the amount to be withdrawn.
34        */
35
36      public void withdraw( int amount )
37      {
38           balance = balance - amount;
39      }
40
41      /**
42        * Deposit the amount requested.
43        *
44        * @param amount the amount to be deposited.
45        */
46
47      public void deposit( int amount )
48      {
49           balance = balance + amount;
```

```
50      }
51
52      /**
53        * The current account balance.
54        *
55        * @return the current balance.
56        */
57
58      public int getBalance()
59      {
60           return balance;
61      }
62  }
```

BankAccount.java has two purposes. First, it describes the behavior of **BankAccount** objects in language that humans (computer programmers) can read and write. Second, it can be ***compiled*** – translated by a Java ***compiler*** into a form the computer can use when we run the bank simulation. We'll discuss compilation later.

Lines 1–13 in the listing are ***comments*** – text that is useful to the programmers who read and write the class definition but of no interest at all to the compiler. Any text from a **//** to the end of a line and any text between a **/*** and a ***/** is a comment. Every Java source file should begin with comments giving the program's name, the author's name, and a date. The body of the program should contain comments describing the class and its methods and comments at places where some explanation will help programmers understand how the program works.

The class definition proper begins on line 15, which says, naturally enough, that what follows is the definition of the **BankAccount** class. That definition extends through line 62. The curly brackets (**{** and **}**) on lines 16 and 62 serve to define the boundaries of the class definition.

Our class definition begins with a description of the field that will hold the account's balance:

```
private int balance;   // work only in whole dollars
```

Because it is labeled **private**, it is visible only within the boundaries of the class definition. The word **int** says that the field stores integer values.

The rest of the definition describes **BankAccount** behavior: three methods specifying how a **BankAccount** will respond to messages and **BankAccount** objects.

A ***method*** is a named sequence of instructions that specifies what an object does when it receives a message having that name. For example, here is the definition of

withdraw, which tells us what a **BankAccount** will do when it receives a **withdraw** message:

```
36      public void withdraw( int amount )
37      {
38           balance = balance - amount;
39      }
```

Line 36 is the ***method header***. It names the method, makes it available to the **public**, tells us that it returns no value (**void**), and describes the arguments – the information the method needs in order to do its job. This method has one integer argument, **amount**, for the amount we wish to withdraw. The ***method body***, enclosed between the curly braces (**{** and **}**), specifies the response to a **withdraw** message: We subtract the amount to be withdrawn from the balance. The definition of **deposit** is similar.

Here is the code for **getBalance**:

```
58      public int getBalance()
59      {
60          return balance;
61      }
```

The empty parentheses on line 58 tell us that **getBalance** needs no outside information to do its job. The **return balance** statement on line 60 makes the value of the **balance** field available to the object that sent the **getBalance** message.

A ***constructor*** is a special method whose purpose is to create a new object. In Java, the constructor method is the one whose name matches that of the class. Think of it as describing what happens when you place an order at an imaginary **BankAccount** factory.

```
25      public BankAccount( int initialBalance )
26      {
27          this.deposit( initialBalance );
28      }
```

The constructor is told (in its **initialBalance** argument) how much money the account will start with. The body of the constructor sends a **deposit** message to **this** object, the **BankAccount** under construction, asking it to deposit the proper amount in itself.

Our simulation has (one) **Bank** object that creates (two) **BankAccount** objects and sends messages to them to perform account transactions. To see how that is expressed in Java, we look at pieces of code from the description of the **Bank** class in file **Bank.java** (Listing 1-2).

Here is the **Bank** object constructor:

```
41      public Bank( String name )
42      {
43          bankName = name;
44          atm      = new Terminal();
45          account1 = new BankAccount( INITIAL_BALANCE );
46          account2 = new BankAccount( INITIAL_BALANCE );
47      }
```

This code shows that whenever we construct a **Bank** object we first set the **bankName** field in the **Bank** to the value passed as an argument. Then we use the Java keyword **new** to create the **Terminal** [7] object **atm** and the two **BankAccount** objects **account1** and **account2**, starting each out with 200, the value of **INITIAL_BALANCE** set on line 29:

```
29      private static final int INITIAL_BALANCE = 200;
```

In our simulation it's line 140 that actually creates the single **Bank** object we interact with. That line tells the new bank its name:

```
140     Bank javaBank = new Bank( "Engulf and Devour" );
```

Message Passing

We create objects in order to make them work for us. Right after we create the bank, we ask it to open itself:

```
141     javaBank.open();
```

That line illustrates message passing syntax: We name the object, then the method, then the information the object needs to do what we are asking – in this case there is no extra information, so the parentheses are empty.

There are many more examples in the code in the **Bank** class that processes transactions for an account:

```
107         String command = atm.readWord( "transaction: " );
108         if ( command.equals( "exit" ) ) {
109             moreTransactions = false;
110         }
111         else if ( command.equals( "help" ) ) {
112             atm.println( HELPSTRING );
113         }
```

[7] **Terminal** objects are described in the file **Terminal.java**, which is Listing 8-1. We won't study that code until Chapter 8. Until then, we will just use **Terminal** objects in our programs.

```
114       else if ( command.equals( "deposit" ) ) {
115           int amount = atm.readInt( "amount: " );
116           account.deposit( amount );
117       }
118       else if ( command.equals( "withdraw" ) ) {
119           int amount = atm.readInt( "amount: " );
120           account.withdraw( amount );
121       }
122       else if ( command.equals( "balance" ) ) {
123           atm.println( account.getBalance() );
124       }
125       else{
126           atm.println( "sorry, unknown transaction" );
127       }
```

Line 107 sends the **atm** (a **Terminal** object) a **readWord** message, telling it what prompt to use when it asks for input from the user. The **Terminal**'s **readWord** method collects the user's input and returns it to the **bank**, which stores the value in the variable **command**. The remaining code (lines 108–27) expresses the logic that examines the value of **command** in order to decide what to do. Read those lines this way:

- The statement **command.equals("exit")** sends the **command** string an **equals** message, asking it if its value (the string the user typed) is "exit". The **equals** method returns **true** or **false**, depending on the answer to the question. If the user did in fact type "exit", set the variable **moreTransactions** to **false** to indicate we're done with this particular bank account.
- Otherwise,[8] if the user typed "help", send the **atm** a message asking it to print out the value of **HELPSTRING**, the **String** listing all possible commands:

```
30   private static final String HELPSTRING =
31      "Transactions: exit, help, deposit, withdraw, balance";
```

- Otherwise, if the user typed "deposit", send the **atm** a **readInt** message, asking it to prompt for, read, and return an integer amount. Then send a **deposit** message to the **BankAccount** object **account**, with that amount as an argument.
- Otherwise, if the user typed "withdraw", send the **atm** a **readInt** message, asking it to prompt for, read, and return an integer amount. Then send a **withdraw** message to the **BankAccount** object **account**, with that amount as an argument.
- Otherwise, if the user typed "balance", send the **atm** a **println** message, asking that it display the current balance. To get that value we first send the account a

[8] In Java, **else** can always be read as "otherwise."

getBalance message. The value returned by **account.getBalance()** is in turn the argument to the **println** message sent to the **atm**.

- Otherwise, the command is not one of the available transactions. Send the **atm** a **println** message, asking it to inform the user.

Compile/Run

In order to run the bank simulation application, we must first ***compile*** all of the files to translate the Java source code into a form the computer can understand. How you tell your computer to compile a file depends on your local programming environment. In this book we assume that environment provides you with a command-line interface that prompts you for what you want do next. Windows usually uses a > as its command prompt; Unix and Linux use a % unless someone has programmed a different prompt. To avoid taking sides in a political controversy, we will use both, writing

```
%> javac BankAccount.java
```

when you are to type "javac BankAccount.java" at the command prompt. This particular command causes the Java compiler to produce a ***class file*** **BankAccount.class**. In general, the class file's name is the same as the class name with a **.class** extension replacing the **.java** extension.

You can compile both files at once with the command

```
%> javac BankAccount.java Bank.java
```

Our banking system requires several more class files in addition to **Bank.class** and **BankAccount.class**. One is **Terminal.class**, containing the description of **Terminal** objects. Another is **String.class**, containing the description of **String** objects. You do not need to do anything special to use the **String** class, which comes along with Java. The **Terminal** class is particular to *Java Outside In*.[9]

When you have created class files by compiling Java source code with the javac compiler you are ready to run the bank simulation. The ***Java Virtual Machine (JVM)*** is a computer program that reads class files and executes their contents. To start the JVM and ask it to start the bank application, we execute the command

```
%> java Bank
```

This produces the behavior we have sketched previously, allowing a user to interact with the program.

[9] If you are using *Java Outside In* as a text in a programming course your instructor will have made the **Terminal** class available to you. If you are studying on your own, consult the section on the *Java Outside In* CD-ROM or Web page `www.cs.umb.edu/joi` to find out how to configure your environment so that your programs can see the **Terminal** class.

We encourage you to look through **`Bank.java`** and **`BankAccount.java`** now, even though this banking application is a larger program (almost 200 lines) than you will usually find in the first chapter of an introductory text. We think it is valuable to encounter a program of this size right at the start of your studies: It's large enough to be genuinely interesting. Just understand that you need not understand it all right now.[10]

A Traffic Light Program

Our second example is an application that simulates the behavior of a traffic light. Following the pattern we have just introduced, we consider first our program's behavior (its user interface), then the objects that model an implementation of that behavior, then part of the Java program that gives life to our light.

Traffic lights at an intersection change color from time to time in order to guarantee a smooth and safe flow of traffic. In the real world, traffic lights are controlled by sophisticated software that responds to input from timers and sensors and buttons. Our program models an unrealistically simple situation – just one light, whose color changes when the user presses just one button.

The Behavior of the Traffic Light Program

We communicated with the bank application using a ***command-line interface***: a sequence of typed commands and typed responses. The traffic light application has a ***graphical user interface*** (GUI): We use the mouse to talk to the program. In response, the program changes the pictures on the screen. GUI programming is more difficult than command-line interface programming. Because we don't think mastering those difficulties is the best way to learn to program, we won't do much of it in this text, but we want to give you a taste of it here in the first chapter.

To run the traffic light application, we compile its source code to build class files:

```
%> javac TrafficLight.java Lens.java Sequencer.java
%> javac NextButtonListener.java
```

and then start the JVM:

```
%> java TrafficLight
```

Figure 1-5 shows a black and white rendering of what you will see on your computer screen: a traffic light showing a green lens lit (the red and yellow lenses are there but invisible), and, beneath the light, a single button marked "Next". When you move your mouse to that button and click it, the light responds by advancing to its next state, from green to yellow, from yellow to red, from red back to green, and so on.

[10] It won't be on the test . . .

Figure 1-5: A traffic light system (green)

How you shut the program down depends on the particular window system your computer is using. You should find both a button to click on and a Close choice on a menu. Either will do.[11]

An Object Model for the TrafficLight Program

The design we have chosen for our application uses several objects. Some of them are visible:

- A **`TrafficLight`** object, with three **`Lens`** objects
- A **`Button`** object to push

Some of them are invisible, but necessary:

- A listener object to respond to clicks on the button
- A **`Sequencer`** object to control the light in response to messages from the button listener.

Figure 1-6 shows the object model for our traffic light application. We can view that picture as a hardware diagram, in which the arrows represent wires that carry electrical signals from one component to the next. We can also see it as an object diagram like Figure 1-4, in which the arrows represent the values of the various objects' fields by pointing to the objects they contain. Some (unspecified) field in the **`Button`** has as its value a **`NextButtonListener`**. The **`sequencer`** field in the **`NextButtonListener`** has as its value a **`Sequencer`**. The **`Sequencer`**'s **`light`** field has as its value a **`TrafficLight`**. The **`TrafficLight`** has three fields named **`red`**, **`yellow`**, and

[11] Recall that whenever you begin working with a new user interface one of your first tasks should be to learn how to shut it down.

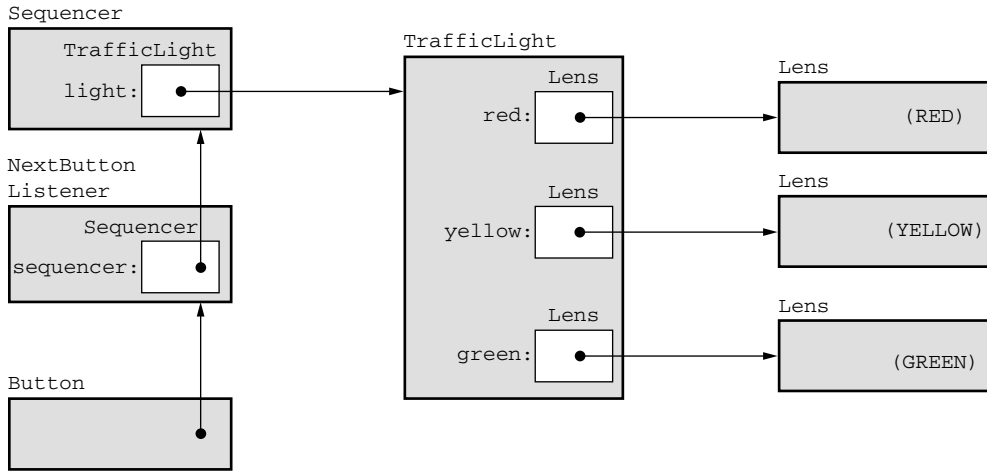

Figure 1-6: Object model for the traffic light application

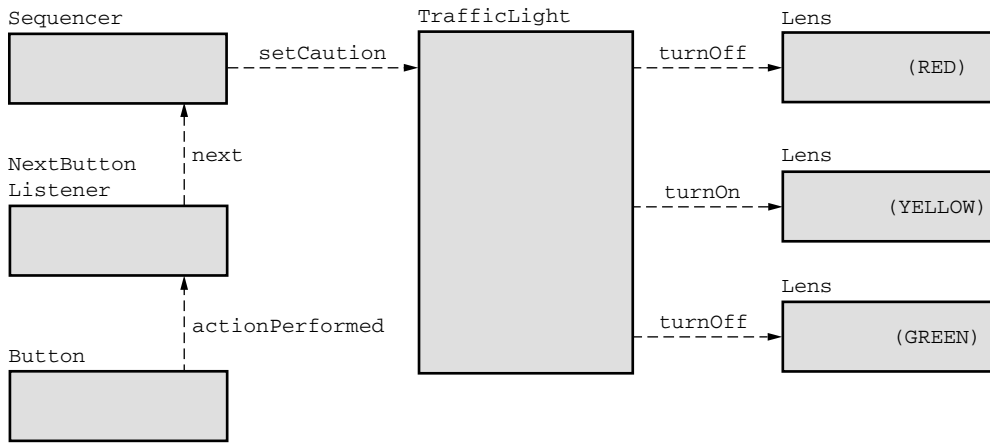

Figure 1-7: Message passing diagram for the traffic light application

`green`, each of which holds a distinct **`Lens`**. Each **`Lens`** has a field that holds its **`color`**.

Recall that in object-oriented programs, work is done when objects send messages to one another. Figure 1-7 represents the messages that will be sent when the light is green and the user presses the "Next" button, represented by the **`Button`** object in our model. The **`Button`** object sends an **`actionPerformed`** message to the **`NextButtonListener`**, which in turn sends a **`next`** message to the **`Sequencer`** object. The **`Sequencer`** responds by shifting itself to the next state and sending the

appropriate message, **setCaution**, to the **TrafficLight**. The **TrafficLight** in turn interprets the **setCaution** message from the **Sequencer** by sending messages to the three lenses: **turnOff** to **red**,[12] **turnOn** to **yellow**, and **turnOff** to **green**.

That changes the light from green to yellow; Figure 1-8 shows the result.

The Traffic Light Application in Java

You will find the source code for the four classes we have written in Listings 1-3, 1-4, 1-5, and 1-6: **TrafficLight.java**, **NextButtonListener.java**, **Sequencer.java**, and **Lens.java**, respectively. We don't need source code for **Button** because **Button.class** (like **String.class**) comes with Java.

Let's take a look at the **Sequencer** class, defined in **Sequencer.java** (Listing 1-5). A **Sequencer** object controls the sequence of states that the traffic light passes through.

A **Sequencer** object has two fields. The **TrafficLight** field, **light**, declared on line 19 refers to the light that this **Sequencer** is controlling. The integer field **currentState**, declared on line 26, holds the state that the traffic light is currently in: **GO, CAUTION**, or **STOP.** The integer constants declared and initialized on lines 22, 23, and 24 represent these states.

A **Sequencer** has just two methods: a constructor, defined on lines 34–9, and the method **next**, defined on lines 47–69.

The constructor initializes the value of the **light** field with the **TrafficLight** passed as its argument and then sets the initial state of the **Sequencer** to **GO** by setting the **currentState** field (line 37) and sending the light a **setGo** message (line 38). (The light responds to this message by running its **setGo** method. If you look

Figure 1-8: A traffic light system (yellow)

[12] The red lens is actually already off at this moment, but this defensive program makes sure by sending the redundant message.

at **TrafficLight.java** you can see how that method sends messages to turn the light's **green** lens on and its **red** and **yellow** lenses off.)

Figure 1-7 shows that the method **next** is invoked in response to **next** messages sent from the **NextButtonListener** object. It uses a **switch** statement to examine the value of the **currentState** field and act appropriately, changing the value of that field and sending the light the right message.

For example, when **currentState** is **GO**, execution of the program branches to the code

```
51      case GO:
52          this.currentState = CAUTION;
53          this.light.setCaution();
54          break;
```

The state is changed to **CAUTION**, and a **setCaution** message is sent to the light (causing it to send appropriate messages to its lenses). The **break** statement on line 54 tells Java that the **switch** on line 49 is done, completing the method at line 69.

Control branches to the **default** clause on line 66 only if **currentState** matches none of the intervening cases. This will not happen in our program.[13]

Thus the **Sequencer** object contains all of the logic necessary for maintaining state and shifting through the states **GO, CAUTION**, and **STOP**, and back to **GO** upon receiving successive **next** messages. We leave the provision of an additional **WALK** state as an exercise.

The Software Development Cycle

In the two examples we have studied, one application turned our computer into an ATM connected to a bank, the other into a traffic light. The process that produced these examples is typical of the way software is developed. That process has several identifiable stages:

1. *Imagine.* You have an idea – an image of the application you wish existed, the special-purpose machine you want to run on your general-purpose computer.
2. *Design.* Think about your application's user interface – how it will behave – and then think about the software objects you will build to implement that behavior.
3. *Edit.* Write the Java code that constructs the objects you have designed.
4. *Compile.* Invoke the Java compiler to construct class files from your source files. If there are any errors, return to Step 3.

[13] A programmer's last words .... Note that our program is designed to print an error message even in a situation we are sure can never occur. That's defensive programming.